**UNIVERSIDADE FEDERAL DE SANTA CATARINA**
**DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA**

Sheiny Fabre Almeida

# PARALELISMO EM ANÁLISE DE *TIMING* ESTÁTICA

Florianópolis

2016

Sheiny Fabre Almeida

# PARALELISMO EM ANÁLISE DE *TIMING* ESTÁTICA

Trabalho de Conclusão de Curso submetido ao Curso de Bacharelado em Ciências da Computação para a obtenção do Grau de Bacharel em Ciências da Computação.

Orientador: Prof. Dr. Laércio Lima Pilla
Universidade Federal de Santa Catarina
Coorientador: Prof. Dr. José Luís Almada Güntzel
Universidade Federal de Santa Catarina

Florianópolis

2016

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Almeida, Sheiny Fabre
Paralelismo em Análise de Timing Estática / Sheiny Fabre Almeida ; orientador, Laércio Lima Pilla ; coorientador, José Luís Almada Güntzel. - Florianópolis, SC, 2016.
96 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico. Graduação em Ciências da Computação.

Inclui referências

1. Ciências da Computação. 2. Análise de Timing Estática. 3. Projeto Físico de Circuitos Integrados. 4. Electronic Design Automation. 5. Paralelismo. I. Lima Pilla, Laércio. II. Almada Güntzel, José Luís. III. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. IV. Título.

Sheiny Fabre Almeida

**PARALELISMO EM ANÁLISE DE *TIMING* ESTÁTICA**

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de "Bacharel em Ciências da Computação", e aprovado em sua forma final pelo Curso de Bacharelado em Ciências da Computação.

Florianópolis, 08 de Julho 2016.

---

Prof. Dr. Renato Cislaghi
Universidade Federal de Santa Catarina
Coordenador

**Banca Examinadora:**

---

Prof. Dr. Laércio Lima Pilla
Universidade Federal de Santa Catarina
Orientador

---

Prof. Dr. José Luís Almada Güntzel
Universidade Federal de Santa Catarina
Coorientador

---

Prof. Dr. Márcio Bastos Castro
Universidade Federal de Santa Catarina

À minha família.

## AGRADECIMENTOS

À minha mãe, Zilda por todo amor incondicional, ao meu pai, Francisco que sempre me apoiou em todas decisões e ao meu eterno melhor amigo e também irmão, Michel.

Agradeço à minha namorada Alessandra pelo amor, compreensão e companheirismo, por estar presente desde o início de minha graduação.

Aos meus orientadores Laércio Lima Pilla e José Luís Almada Güntzel, pela amizade, por toda confiança, dedicação e todo aprendizado proporcionado. Por terem me guiado pelo caminho do conhecimento.

Ao professor Márcio Bastos Castro, pela participação na avaliação deste trabalho, pelas aulas ministradas durante a graduação que foram fundamentais para produção deste trabalho e também minha escolha sobre área de pesquisa.

Aos colegas da graduação Rodrigo P. de Mello, Luiz De L. Cancellier, Luis Decker, Eduardo Beckhauser, Tiago Fontana, Márcio Monteiro e demais colegas do ECL que de alguma forma participaram deste trabalho.

*People have forgotten this truth,"the fox said. "But you mustn't forget it. You become responsible forever for what you've tamed. You're responsible for your rose.*

(Antoine de Saint-Exupéry)

## RESUMO

O presente trabalho visa estudar o uso de programação paralela aplicada à Análise de *Timing* Estática - *Static Timing Analysis* (STA), uma técnica utilizada para estimar o atraso de um circuito durante a etapa de síntese física. O estudo envolve a identificação de oportunidades de paralelismo e possíveis melhorias na estrutura de dados para aprimorar a solução paralela. Por fim, o trabalho inclui o desenvolvimento de soluções paralelas com base no ferramental desenvolvido no Laboratório de Computação Embarcada (ECL) da Universidade Federal de Santa Catarina (UFSC).



**Palavras-chave:** Análise de Timing Estática, Projeto Físico de Circuitos Integrados, Electronic Design Automation, Paralelismo

## ABSTRACT

This work seeks to study the use of parallel programming applied to Static *Timing* Analysis (STA), which is a technique used to estimate the expected delay of a circuit during the physical synthesis step. The study involves the identification of parallel opportunities and possible improvements in the data structure in order to improve the parallel solution· Lastly, this work includes the development of parallel solutions based on toolkit developed in Embedded Computing Laboratory (ECL) from Federal University of Santa Catarina (UFSC).



**Keywords:** Static Timing Analysis, Physical Design of Integrated Circuits, Electronic Design Automation, Parallelism

## LISTA DE FIGURAS

## LISTA DE SIGLAS E ABREVIATURAS

## SUMÁRIO

# 1 INTRODUÇÃO

Este capítulo tem por objetivo apresentar uma visão geral sobre o fluxo de projeto e a importância da análise de *timing* para o desenvolvimento de Circuitos Integrados - *Integrated Circuits* (CIs). Serão apresentados também a motivação, justificativa e objetivos deste trabalho.

## 1.1 PROJETO DE CIRCUITOS DIGITAIS

Os CIs constituem o núcleo de qualquer equipamento eletrônico contemporâneo. O projeto de CIs requer uma sequência de etapas, dentre as quais a síntese física é a responsável por gerar a descrição fabricável. A síntese física busca posicionar os *layouts* das portas lógicas e *flip-flops* (referenciados por "células") sobre uma região 2-D, realizando as conexões necessárias entre estes e entre os *flip-flops* e o sinal de relógio (KAHNG et al., 2011, p. 8).

O posicionamento de células resultará em uma configuração topológica com um determinado atraso. A fim de satisfazer as restrições de atraso do circuito para atingir a frequência de relógio alvo, diversas técnicas de otimização são aplicadas na síntese física. Dentre tais técnicas citam-se *gate sizing*, inserção de *buffers* e Posicionamento Incremental Guiado por Atraso - *Incremental Timing Driven Placement* (ITDP) (KAHNG et al., 2011, p. 221).

Todas essas técnicas necessitam de estimativas precisas de atraso do circuito, a fim de guiar a otimização e garantir a convergência das técnicas empregadas. Além disso, tal estimativa deve ser obtida com o menor tempo de execução possível, pois ela precisa ser feita a cada iteração da otimização. A Análise de *Timing* Estática - *Static Timing Analysis* (STA) é utilizada para tal propósito. Por questões de eficiência computacional, a STA é um método utilizado para computar o atraso esperado de um circuito digital sem realizar qualquer tipo de simulação (KAHNG et al., 2011, p. 222).

### 1.1.1 Fluxo de Projeto

O projeto de um CI é um processo que envolve uma sequência básica de etapas, partindo de uma descrição comportamental em alto

nível, até o encapsulamento e teste do CI. Esse fluxo de síntese é ilustrado na Figura 1.

A realização do fluxo de projeto completo demanda um grande esforço computacional por parte das ferramentas de Automação de Projeto Eletrônico - *Electronic Design Automation* (EDA) durante as diversas otimizações que são realizadas, como por exemplo as que foram citadas anteriormente.

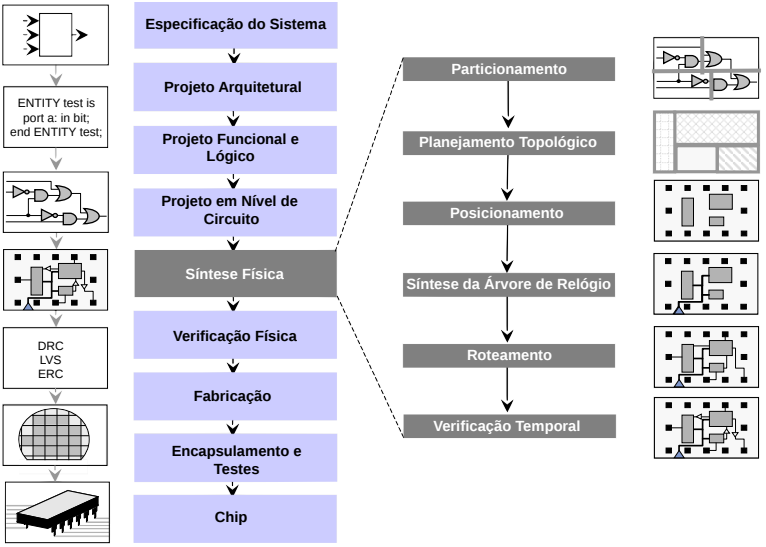



Figura 1 – Principais etapas do fluxo de síntese de um circuito integrado, *Very-Large-Scale Integration* (VLSI). Adaptado de (KAHNG et al., 2011, p. 7).

Na etapa de **Especificação do Sistema** é descrito o funcionamento do sistema, quais são as entradas e saídas que o circuito deve produzir.

Na etapa denominada **Projeto Arquitetural** são tomadas as decisões arquiteturais sobre o projeto, por exemplo: se blocos de propriedade intelectual serão utilizados, qual será o tipo de memória utilizada, como será feito o gerenciamento da memória (serial ou paralelo) e até mesmo qual processo tecnológico será utilizado.

Com a arquitetura do projeto já definida pela etapa anterior, na etapa de **Projeto Funcional e Lógico** a funcionalidade e conectividade dos módulos precisa ser definida.

Durante o projeto funcional o comportamento em alto nível de abstração precisa ser determinado, ou seja, cada módulo possui um conjunto de entradas, saídas e um comportamento temporal.

O projeto lógico é feito em Nível de Transferência entre Registradores - *Register Transfer Level* (RTL) utilizando uma Linguagem de Descrição de *Hardware* - *Hardware Description Language* (HDL), como VHDL ou *Verilog*. Essas linguagens descrevem o funcionamento do circuito e um comportamento temporal. No entanto, esses módulos descritos em HDL precisam ser simulados e verificados.

***Projeto em Nível de Circuito***: Ferramentas de síntese física são utilizadas para automatizar a conversão da HDL em uma descrição em baixo nível, essas ferramentas realizam a conversão utilizando a HDL juntamente com uma biblioteca tecnológica. Apesar disso, alguns elementos críticos do circuito precisam ser projetados a nível de transistor, isto é chamado de *Circuit Design*. Alguns destes elementos críticos podem ser: blocos de memória, unidades lógica e aritmética de alto desempenho, entrada/saída, etc.

O presente trabalho está fortemente relacionado à etapa de **Síntese Física**, uma vez que a técnica abordada é essencial para diversos passos de otimização (por exemplo: dimensionamento/seleção de portas, posicionamento etc.). Além disso, tal configuração física deve otimizar o consumo energético e atraso de modo a satisfazer às restrições do projeto. Mais detalhes sobre essa etapa são discutidos na próxima subseção.

Por fim, as etapas seguintes têm como objetivo testar e validar o circuito. Caso o mesmo cumpra com os requisitos especificados pelo projeto, são produzidas as máscaras necessárias para fotolitografia, possibilitando a produção em larga escala do circuito.

### 1.1.2 Síntese Física

A síntese física tem como objetivo principal gerar uma descrição geométrica para o circuito que é utilizada para fabricar as máscaras necessárias para a fotolitografia (KAHNG et al., 2011, p. 8). Tal descrição é composta por retângulos que descrevem as etapas necessárias para a fabricação dos transistores e dos fios necessários para formar as portas lógicas e as conexões entre estas. Para reduzir a complexidade da síntese física, os layouts dos transistores e dos fios usados para formar as portas lógicas e os elementos sequenciais (*latches* e *flip-flops*), referenciados por células, são pré-projetados e disponibilizados sob a forma

de biblioteca de células. As bibliotecas são fornecidas pelas fábricas de circuitos integrados, chamadas de *silicon foundries*, e incluem também tabelas com informações de atrasos e consumo energético de cada célula, obtidas por meio de simulações no nível elétrico. Desta forma, o fluxo de síntese física industrial é baseado no uso de bibliotecas sendo por este motivo também denominado de fluxo *standard cell* (KAHNG et al., 2011, p. 13).

Conforme mencionado na subseção anterior, a solução fornecida pela etapa de síntese física tem forte impacto sobre as características físicas do circuito. Porém, como os CIs podem ter dezenas ou centenas de milhares de portas lógicas, a busca pela solução ótima é inviável, dadas as restrições de tempo para que o produto chegue ao mercado. Isso acontece porque, à medida que a complexidade do circuito aumenta, o esforço para encontrar a melhor solução possível cresce exponencialmente, tornando assim o problema de otimização intratável (KAHNG et al., 2011, p. 224).

Devido à alta complexidade do projeto de um CI, o uso de uma ferramenta de análise de *timing* rápida e precisa é essencial para que seja possível encontrar uma descrição física fabricável que atenda aos requisitos de desempenho impostos ao projeto. (KAHNG et al., 2011, p. 221).

## 1.2 MOTIVAÇÃO

Dada a complexidade crescente dos projetos de CIs, a verificação temporal vem se tornando rapidamente uma forte limitadora para a síntese física (HUANG et al., 2016). O algoritmo mais conhecido e trivial para STA é a análise de *timing* topológica, onde o circuito é representado como um Grafo Acíclico Direcionado - *Direct Acyclic Graph* (DAG) e é percorrido em ordem topológica, propagando os atrasos dos elementos de maneira cumulativa, das entradas para as saídas do circuito (HU; SCHAEFFER; GARG, 2015). **Tendo em vista que a análise de *timing* topológica não apresenta paralelismo em sua forma padrão, este trabalho propõe um estudo sobre técnicas de paralelismo que possam ser empregadas na análise de *timing* estática.**

## 1.3 OBJETIVOS GERAIS

O objetivo deste trabalho é propor, implementar e validar soluções paralelas para a STA tomando como base a ferramenta de análise de *timing* já desenvolvida pelo Laboratório de Computação Embarcada - ECL, laboratório onde pesquisas no tema de síntese física são desenvolvidas (ECLUFSC, 2016).

### 1.3.1 Objetivos Específicos

- Implementar uma versão paralela de algoritmo de STA.
- Realizar experimentos com diferentes estratégias de paralelismo para comparar resultados obtidos.
- Verificar a escalabilidade e comportamento das técnicas desenvolvidas quando aplicadas a diferentes arquiteturas computacionais.

## 1.4 ESCOPO

Este trabalho aborda o problema da STA, utilizando técnicas de paralelismo de modo a acelerar a verificação temporal sem comprometer a qualidade dos resultados, tornando assim viável a exploração de diferentes técnicas de otimização que necessitam de um maior tempo de execução.

## 1.5 ORGANIZAÇÃO DO TEXTO

Este trabalho está organizado da seguinte forma:

- O Capítulo 2 apresenta conceitos sobre circuitos, temporização de CIs, os modelos de atraso adotados neste trabalho e um exemplo do cálculo de atraso de um CI. Este capítulo também discute conceitos de programação paralela, a linguagem de programação paralela adotada e principais trabalhos relacionados.
- O Capítulo 3 apresenta o algoritmo de STA em detalhes, a complexidade do algoritmo, dependência dos dados e a estratégia de paralelismo adotada para a resolução do problema.

- O Capítulo 4 apresenta a metodologia de testes adotada e os diferentes ambientes de teste utilizados e também uma análise e discussão a respeito dos resultados obtidos.
- O Capítulo 5 apresenta as conclusões obtidas com este trabalho e perspectivas de trabalhos futuros.

# 2 CONCEITUAÇÃO

O objetivo deste capítulo é explanar alguns conceitos teóricos necessários para a compreensão da presente monografia.

## 2.1 CONCEITOS SOBRE CIRCUITOS INTEGRADOS

- **Rede de Relógio (*Clock Network*):** Uma rede onde um sinal de relógio é propagado de uma fonte para todos os elementos sequenciais. O sinal de relógio é muito importante pois ele é responsável não somente pela sincronia dos elementos sequenciais como também por limitar a frequência de operação do circuito (KAHNG et al., 2011, p. 197).
- **Elementos Sequenciais:** São responsáveis pela sincronização dos sinais do circuito, ou seja, dividem o circuito temporalmente em estágios. Também conhecidos como elementos de armazenamento. Exemplos de elementos sequenciais são *flip-flops* e *latches*.
- **Elementos Combinacionais:** São responsáveis por realizar operações lógicas e, portanto, pela computação do circuito. Em circuitos digitais, esses elementos combinacionais implementam funções booleanas como *and*, *or*, *xor* e outras.
- **Interconexões:** São as trilhas de metal roteadas internamente no CI por onde os sinais elétricos são propagados.

Cada elemento que compõe o circuito é responsável por introduzir um determinado atraso na propagação do sinal (HU; SCHAEFFER; GARG, 2015). A seguir serão apresentados diferentes tipos de atrasos propagados por tais elementos.

- ***Clock Skew*:** É a diferença máxima na variação dos tempos de chegada do sinal de relógio nas entradas dos elementos sequenciais. Ou seja, dada uma borda de relógio, o *clock skew* será a diferença entre o maior atraso de chegada em um determinado *flip-flop* e o menor atraso em outro *flip-flop* (KAHNG et al., 2011, p. 200). Para todos os exemplos apresentados neste trabalho, o efeito do *clock skew* será desconsiderado.
- ***Delay* (Atraso de Propagação):** Atraso de propagação (ou simplesmente, atraso) é a quantidade de tempo necessária para

que, ao ocorrer uma mudança no sinal de entrada de uma porta lógica ou interconexão, essa seja propagada para a saída da porta ou da interconexão. Se o sinal na saída da porta ou da interconexão for de nível lógico baixo (0) para um nível lógico alto (1) este atraso é chamado de atraso de subida (*rise delay*), caso contrário, é chamado de atraso de descida (*fall delay*). Esse comportamento está ilustrado pela Figura 2, considerando-se uma porta inversora.

- ***Slew* (Tempo de Transição):** Quantidade de tempo que um sinal precisa para, em determinado ponto, transicionar de um nível lógico para outro. Se a transição nesse ponto for de nível lógico baixo (0) para um nível lógico alto (1), esse atraso é chamado de transição de subida (*rise slew*), caso contrário é chamado de transição de descida (*fall slew*). Esse comportamento é ilustrado na Figura 2.

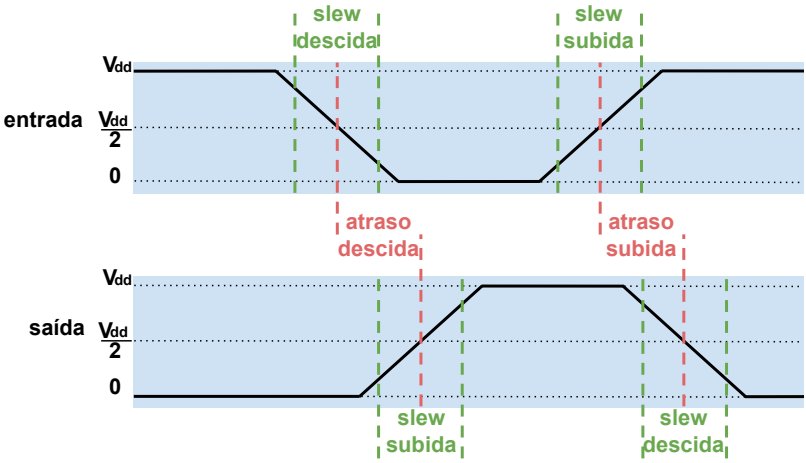
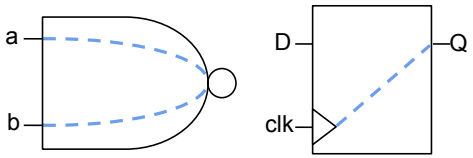



Figura 2 – Uma representação em forma de onda da transição do sinal na entrada (*input*) e na saída (*output*) de uma porta lógica inversora.

- ***Timing Arc* (Arco de Tempo):** O arco de tempo envolve um pino de entrada de elemento do circuito em direção ao pino de saída desse elemento. Esses arcos de tempo são chamados de *Positive Unate* caso uma transição do sinal na entrada do elemento faça com que ocorra uma transição do sinal no mesmo sentido na saída do elemento, caso contrário será chamado de

*Negative Unate*. A Figura 3 possui exemplos de *timing arcs*.




(a) (b)

Figura 3 – (a) Uma porta lógica *NAND* com seus dois arcos de tempo representados pelas duas linhas tracejadas. (b) Um *flip-flop* com seu arco de tempo entre o sinal de relógio e a saída representado pela linha tracejada.

## 2.2 ANÁLISE DE *TIMING* ESTÁTICA

O comportamento de um circuito sequencial pode ser representado pela Figura 4, onde os elementos sequenciais (retângulos) dividem o circuito temporalmente em vários estágios.

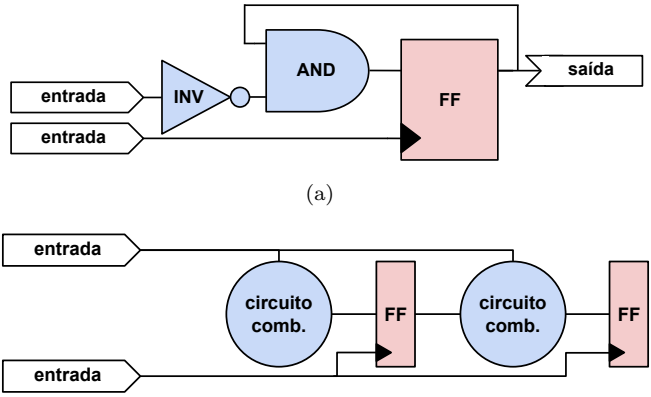



Figura 4 – (a) Uma ilustração de um circuito sequencial. (b) O comportamento temporal do mesmo circuito ilustrado em (a) desenrolado no tempo. Adaptado de (KAHNG et al., 2011, p. 223).

No início de cada ciclo de relógio, os sinais são propagados a partir das entradas ou saídas dos elementos sequenciais. Esses sinais percorrem os elementos combinacionais do circuito e, mais tarde, são capturados pelos elementos sequenciais ou amostrados nas saídas do circuito. Esse comportamento se repete enquanto o circuito estiver em funcionamento.

A STA é um método que calcula o tempo estimado que um sinal elétrico leva para estabilizar em um dos dois níveis válidos (0 V ou Vdd) em um determinado ponto do circuito. Essas estimativas de tempo servem para verificar se existe algum tipo de violação temporal no circuito em relação às restrições de projeto estabelecidas.

A funcionalidade lógica do circuito é desconsiderada pela STA ou seja, cada pino é capaz de propagar uma transição de subida ou descida. Os atrasos são propagados de maneira cumulativa a partir das entradas para as saídas do circuito (KAHNG et al., 2011, p. 222).

A palavra estática significa que toda a análise pode ser feita de forma independente dos valores lógicos dos sinais das entradas do circuito. O principal propósito dessa técnica é encontrar o pior caso de atraso do circuito dentre todas as possíveis combinações para os sinais de entrada (SAPATNEKAR, 2004, p. 99).

O modelo computacional mais comum para representar um circuito eletrônico é através de um DAG (KAHNG et al., 2011, p. 224).

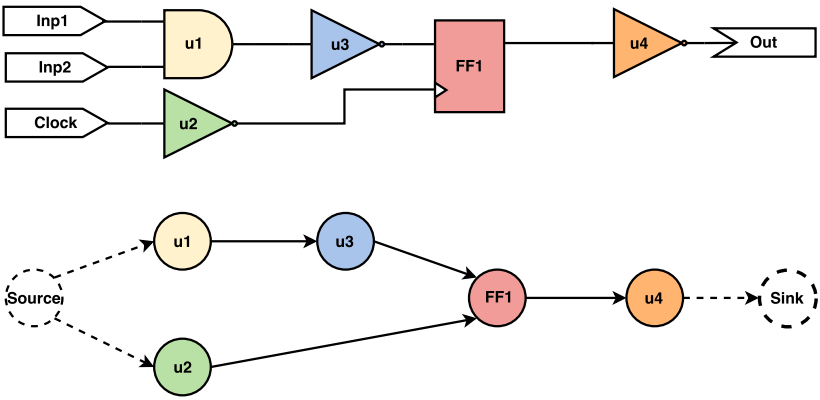



Figura 5 – Uma possível representação de um circuito através de um DAG.

Na Figura 5, os vértices representam portas lógicas e as arestas representam as interconexões entre as portas. Também foram adici-

onados dois vértices, um fonte (*source*) e um sumidouro (*sink*), para representar as entradas e saídas do circuito, respectivamente. Devido à presença de um elemento sequencial no circuito, no caso o *flip-flop* (FF1), o circuito é dividido temporalmente, formando assim dois subcircuitos (Figura 6).

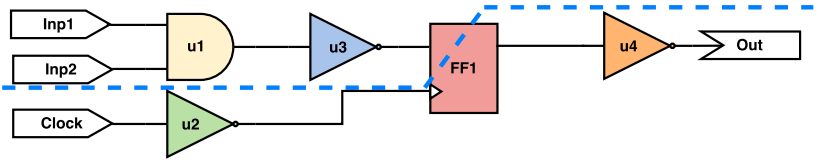



Figura 6 – Dois subcircuitos separados temporalmente por um elemento sequencial (*flip-flop* FF1).

A propagação do atraso é feita a partir das entradas do circuito até as saídas. Com o objetivo de determinar o correto funcionamento do circuito, tempos de chegada e verificação são estabelecidos. Esses conceitos são descritos abaixo.

- **Pontos Iniciais de *Timing* - *Timing Startpoints* (TS):** Pontos de partida para o cálculo do atraso. Esses pontos pertencem ao conjunto formado pelas entradas primárias do circuito união com as saídas dos elementos sequenciais.
- **Pontos Terminais de *Timing* - *Timing Endpoints* (TE):** Pontos de término para o cálculo do atraso. Esses pontos pertencem ao conjunto formado pelas saídas primárias do circuito união com as entradas dos elementos sequenciais.
- **Tempo de Chegada - *Arrival Time* (AT):** Cada ponto do circuito terá um valor de atraso associado. Esse atraso é calculado partindo dos TSs do circuito até os TEs, sendo os atrasos propagados de forma cumulativa.
- **Tempo de Chegada Requerido - *Required Arrival Time* (RAT):** Cada ponto do circuito é associada uma restrição de atraso. Essas restrições refletem as especificações fornecidas pelo projeto para o correto funcionamento do circuito. O RAT é calculado partindo dos TEs do circuito até os TSs, descontando os atrasos gerados pelos elementos do circuito.
- **Tempo de Folga - *Slack*:** Dado RAT e AT em determinado ponto do circuito, é possível calcular a folga (*Slack*). O *slack* é útil

> para verificar quais caminhos precisam ser otimizados para satisfazer às restrições de atraso. Caso o *slack* seja positivo, o caminho pode ser relaxado de modo a otimizar o consumo energético. Caso contrário, o caminho precisa ser acelerado pois faz parte do caminho crítico (KAHNG et al., 2011, p. 222).

O cálculo do atraso é feito seguindo um método conhecido como Método do Caminho Crítico - *Critical Path Method* (CPM) (SAPATNEKAR, 2004, p. 99). A seguir será demonstrada a forma de calcular os atrasos para o pior caso, ou seja, o atraso pessimista (*late delay*). O cálculo do atraso otimista (*early delay*) segue um método análogo.

O AT é calculado pela Equação 2.1, onde $a_i$ é o AT no nó $i$, $j$ é um antecessor ao nó $i$, $a_j$ o AT em $j$ e $d_{j \to i}$ o atraso de propagação entre $j$ e $i$.

$$a_i = \max_{\forall antecessor\ v_j} (a_j + d_{j \to i}) \tag{2.1}$$

O *slew* é calculado pela Equação 2.2, onde $s_i$ é o *slew* no nó $i$, $j$ é um antecessor ao nó $i$, $s_j$ o *slew* em $j$ e $d_{j \to i}$ o atraso de propagação do *slews* entre $j$ e $i$.

$$s_i = \max_{\forall antecessor\ v_j} (s_j + d_{j \to i}) \tag{2.2}$$

O RAT é calculado pela Equação 2.3, onde $r_i$ é o RAT no nó $i$, $j$ é um sucessor ao nó $i$, $r_j$ o RAT em $j$ e $d_{j \to i}$ o atraso de propagação entre $j$ e $i$.

$$r_i = \min_{\forall sucessores\ v_j} (r_j - d_{i \to j}) \tag{2.3}$$

O *slack* é calculado pela Equação 2.4, onde $slack_i$ é o *slack* no nó $i$, $r_i$ é o RAT nó $i$ e $a_i$ o AT no nó $i$.

$$slack_i = r_i - a_i \tag{2.4}$$

As estimativas de tempo que são calculadas pela STA são os tempos mais cedo e mais tarde que um sinal leva para estabilizar.

A Figura 7 ilustra as verificações temporais de *hold* e *setup*, relacionando os conceitos de atraso mencionados anteriormente. Essas verificações temporais servem para garantir que o atraso vai satisfazer o pior e o melhor caso. O atraso depende de diversos fatores, dentre os quais por exemplo, das condições de operação do circuito e do caminho tomado pelo sinal.

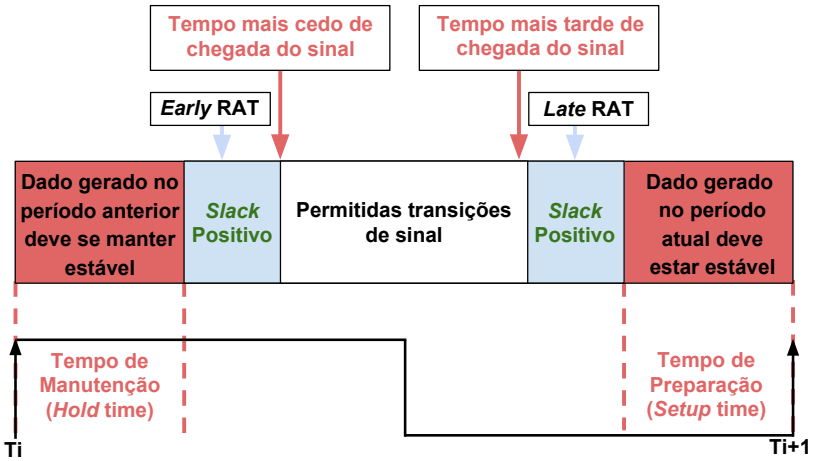



Figura 7 – Restrições de *Hold* e *Setup* ilustradas em um ciclo de relógio, onde $Ti$ é a última borda de relógio e $Ti+1$ a próxima borda. Adaptado de (VISWESWARIAH, 2007, p. 6).

- ***Setup* (Tempo de Preparação):** Uma restrição que especifica o tempo máximo que um sinal, na entrada de um elemento sequencial, deve estar estável antes da borda de relógio. Também conhecida como restrição de caminho longo (KAHNG et al., 2011, p. 221). Uma violação do tempo de *setup* representa o caso onde o sinal chega muito tarde na entrada de um *flip-flop* sem tempo suficiente para estabilizar, consequentemente o sinal pode ser capturado de forma errada. A restrição de *setup* é calculada, com o objetivo de acomodar o **pior caso** para a propagação dos atrasos e portanto, utiliza os valores pessimistas dos atrasos (*late*). Durante a verificação temporal é necessário que para cada elemento sequencial, a Equação 2.5 seja respeitada.

$$t_{ciclo} \geq tc_{max} + t_{setup} + t_{skew} \tag{2.5}$$

Na Equação 2.5, $t_{ciclo}$ representa o período de relógio, que consequentemente vai limitar a frequência máxima de operação do chip. O $tc_{max}$ é o tempo **máximo** necessário para propagar o sinal através do bloco combinacional. O $t_{setup}$ é o tempo necessário para que o elemento sequencial possa capturar corretamente o si-

nal. O $t_{skew}$ serve para acomodar as variações de atraso no sinal de relógio, sendo porém desconsiderado neste trabalho.

- ***Hold* (Tempo de Manutenção):** Uma restrição que especifica o tempo mínimo que um sinal de entrada deve estar estável após a borda de relógio para cada elemento sequencial. Também conhecida como restrição de caminho curto (KAHNG et al., 2011, p. 221). As restrições de *hold* tem como objetivo garantir que não ocorra uma transição de sinal muito cedo. Em um caminho muito curto o sinal pode percorrer rapidamente o bloco combinacional e retornar, ao *flip-flop*, fazendo com que seja capturado o sinal do ciclo atual ao invés do sinal do próximo ciclo.

$$t_{min} \geq t_{hold} + t_{skew} \tag{2.6}$$

A restrição de *hold* é calculada com o objetivo de acomodar o **melhor caso** para a propagação dos atrasos e portanto, utiliza os valores mais otimistas para os atrasos (*early*). A Equação 2.6 representa a restrição de *hold*, onde o $t_{min}$ é o tempo **mínimo** necessário para propagar o sinal através do bloco combinacional e o $t_{hold}$ é o tempo necessário para que o elemento sequencial possa capturar corretamente o sinal. O $t_{skew}$ segue o mesmo motivo explicado na Equação 2.5.

### 2.2.1 Modelos de Atraso Adotados

Esta subseção apresenta os modelos de atraso adotados no presente trabalho para o cálculo do atraso das interconexões e células.

Este trabalho adota o modelo de *Elmore* (*Elmore Delay*) para o cálculo dos atrasos das interconexões (WESTE; HARRIS, 2010, p. 150). Em um CI, uma interconexão pode ser representada por uma Árvore RC (*RC Tree*). A raiz dessa árvore representa o nó fonte e as folhas são as capacitâncias relacionadas aos pinos de saída da interconexão. O *Elmore Delay* estima o atraso dos nós da interconexão modelada como árvore RC quando uma fonte de tensão do tipo degrau (*step*) é aplicada ao seu nó fonte (WESTE; HARRIS, 2010, p. 150). Uma Árvore RC pode ser representada por um DAG, onde os vértices são as capacitâncias e as arestas as resistências entre vértices. Um exemplo de árvore RC e de sua representação são mostradas na Figura 8.

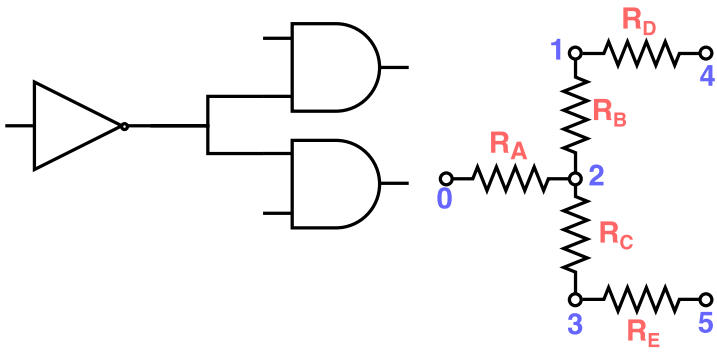



Figura 8 – Uma possível representação para a interconexão entre as portas através de uma Árvore RC.

O cálculo do atraso pode ser feito através do somatório do produto da capacitância acumulada no nó com a resistência efetiva (caminho comum entre o nó folha e a fonte), como representado na Equação 2.7.

$$t_{pd} = \sum_{i} R_i C_i \tag{2.7}$$

Para o caso ilustrado na Figura 8, o atraso de propagação entre o nó fonte e o nó 5 é calculado conforme apresentado na Equação 2.8.

$$d_5 = R_A(C_1 + C_2 + C_3 + C_4 + C_5) + R_C(C_3 + C_5) + R_E C_5 \tag{2.8}$$

A estimativa do atraso de uma célula é feita utilizando uma tabela de atrasos chamada de *lookup table*. Essa tabela é fornecida juntamente com a biblioteca de células. Cada uma dessas tabelas possui informações de atraso referentes a cada *timing arc* da célula e também para cada tipo de transição (subida ou descida). Esses atrasos são definidos em função da capacitância acumulada na saída da porta (linhas) e o *slew* de entrada (colunas). A Figura 9 ilustra uma *lookup table*. Caso os valores da tabela não correspondam exatamente com as entradas, uma interpolação se faz necessária.

```
cell_rise (delay_outputslew_template_7X8) {

load (0.0 1.0 2.0 4.0 8.0 16.0 32.0) ;

input_slew (5.0 30.0 50.0 80.0 140.0 200.0 300.0 500.0) ;

values (
    14.06, 21.86, 27.20, 33.13, 41.78, 48.52, 57.85, 73.48,
    20.31, 28.11, 34.3, 41.9, 52.9, 61.41, 72.7, 90.99,
    26.55, 34.35, 40.59, 49.48, 62.67, 72.68, 86.00, 106.,
    39.0, 46.8, 53., 62.4, 79.2, 92.11, 109.0, 134.80,
    64.05, 71.85, 78.09, 87.45, 106.17, 123.7, 146.8, 181.7,
    114.0, 121.8, 128., 137.4, 156.1, 174., 205.90, 255.97,
    214, 221.85, 228.09, 237.45, 256.17, 274, 306.09, 368.49
);
}
```

-

Figura 9 – Uma *lookup table* para atraso de subida (*rise delay*) de um *timing arc* de uma porta inversora.

### 2.2.2 Exemplo de Cálculo de Atraso

Por questões de simplicidade, neste exemplo, será desconsiderando o caso otimista (*early*), ou seja, os atrasos serão calculados considerando somente o pior caso (*late*), como ilustrado pelas Equações 2.1, 2.3 e 2.4. Dadas as restrições de projeto (RAT) para as saídas e os tempos de chegada para as entradas (ATs), o atraso do circuito pode ser estimado.

Para este exemplo, Figura 10, o atraso das interconexões será igual a 2 picossegundos e o atraso dos *timing arcs* das portas de 1 picossegundo. Sendo o AT para as entradas u1:a, u1:b e u2:a igual à 5, 7 e 4 picossegundos respectivamente e RAT de 13 picossegundos na saída.

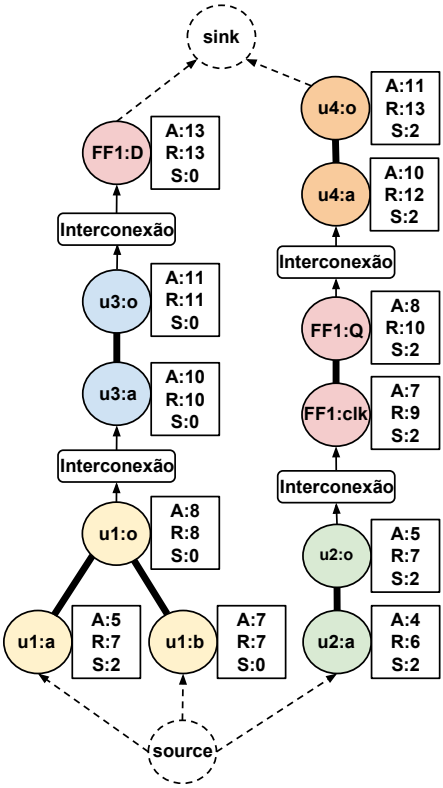



Figura 10 – Uma representação em forma de DAG dos dois subcircuitos ilustrados pela Figura 6. Vértices de mesma cor representam o mesmo elemento lógico. Sendo AT (**A**), RAT (**R**) e *Slack* (**S**).

## 2.3 MULTIPROCESSAMENTO

Arquiteturas computacionais multiprocessadas são caracterizadas por possuir múltiplos núcleos de processamento que compartilham memória. Dessa forma, o processamento dos dados pode ser distribuído entre os núcleos que colaboram para a resolução de um determinado problema.

O acesso à memória em uma arquitetura multiprocessada, pode ser dividido em duas categorias:

- Acesso Uniforme à Memória - *Uniform Memory Access* (UMA): Memória centralizada, encontra-se a mesma distância de todos os processadores, fazendo com que a latência de acesso a memória seja a mesma para todos os processadores. A centralização da memória pode se tornar um gargalo para o sistema, pois pode ser que muitas escritas e leituras concorram para o acesso a mesma.

- Acesso Não Uniforme à Memória - *Non-Uniform Memory Access* (NUMA): Memória é distribuída, implementada utilizando múltiplos nós associados aos processadores. Neste sistema o espaço de endereçamento é único. O tempo de acesso à memória depende do endereço gerado pelo processador: se o endereço se refere ao nodo de memória associado ao próprio processador, a latência de acesso será menor em relação a um endereço de memória associado à um processador remoto. Isto ocorre porque é necessário o uso de uma rede de interconexão para acessar os dados do nodo remoto.

Para utilizar a memória *cache* de maneira eficiente, é importante considerar a afinidade de uma *thread* ou processo. Para isto, é feito um mapeamento da *thread* ou do processo a um determinado núcleo, fazendo com que esses elementos mapeados executem somente no núcleo mapeado, ao invés de executar em qualquer núcleo disponível. A vantagem do mapeamento é que caso o elemento em execução seja posto em espera, ao voltar para o núcleo onde estava sendo executado, possivelmente dados do programa ainda estarão na memória *cache*, evitando a necessidade de novamente buscar pelos dados.

### 2.3.1 Programação Paralela

Em programação paralela um problema é dividido em várias partes, onde essas podem ser computadas concorrentemente. Para entender melhor a execução de um programa paralelo, é importante que fiquem claro os conceitos:

- **Concorrência:** É a condição de um sistema onde múltiplas tarefas estão logicamente ativas em um determinado momento. Isto significa que não necessariamente o programa está executando em paralelo. Isto pode ocorrer por diversos motivos. Por exemplo: pode ser que a dependência de dados limite a execução do programa; o *hardware* pode não fornecer infra estrutura necessária para que execute em paralelo.

- **Paralelismo:** É a condição de um sistema onde tarefas são processadas simultaneamente em diferentes unidades de processamento.

- **Condição de corrida:** A condição de corrida é causada pela concorrência de tarefas que podem executar simultaneamente de forma que o resultado final da execução destas tarefas depende da ordem em que são executadas. Isso se torna um problema quando pelo menos uma dessas tarefas realiza a escrita em um dado compartilhado, fazendo com que outra *thread* leia o valor desatualizado (leitura suja) ou sobreescreva o dado (escrita perdida).

### 2.3.2 *OpenMP*

Para fornecer suporte ao multiprocessamento, neste trabalho utilizou-se a interface de programação paralela *Open Multi-Processing* (OpenMP) (OPENMP, 2015). OpenMP é uma Interface de Programação de Aplicativos - *Application Programing Interface* (API), que fornece uma série de diretivas ao compilador para mapear as tarefas aos núcleos de processamento. Através de memória compartilhada é realiza a sincronização e comunicação entre as tarefas (OPENMP, 2015).

O fluxo de execução paralela fornecido pelo OpenMP segue o modelo *fork-join*, onde *threads* são criadas em uma região paralela e o trabalho é então distribuído, cada uma dessas *threads* executa parte do trabalho de forma concorrente. Conforme ilustrado pela Figura 11.

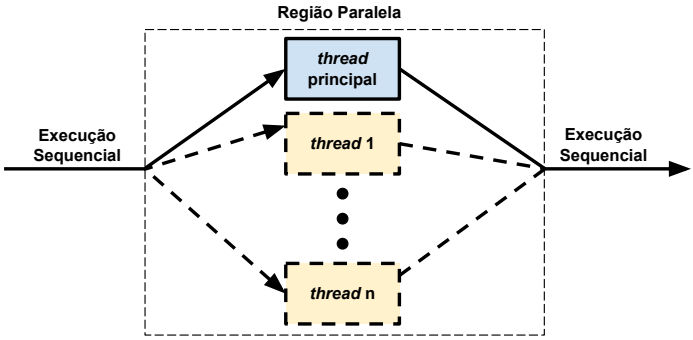



Figura 11 – Uma representação do modelo *fork-join* onde em uma região paralelo o fluxo de execução é dividido entre n *threads*.

Dentre as principais diretivas de compilação, destacam-se:

- **Parallel:** Define uma região paralela, parte do código que será executada por várias *threads*.

- **Shared:** Esta diretiva específica que um dado é compartilhado, existe uma única instância do dado, caso alguma *thread* altere o dado, réplicas do dado serão atualizadas.

- **Private:** Esta diretiva específica que um dado é privado, cada *thread* possui uma cópia do dado, alterações ao dado só serão visíveis no escopo da *thread* que alterou.

- **For:** Utilizada para paralelizar laços, define como serão executadas as iterações paralelas do laço.

- **Task:** Diretiva utilizada para criar uma tarefa, a tarefa é então adicionada em uma lista de tarefas, onde futuramente será executada.

- **Critical:** Especifica que determinada região só será executada por uma única *thread* por vez.

- **Sections:** Utilizado para dividir o código em seções, onde cada seção é executada por uma única *thread*.

- **Single:** Especifica que determinada região do código será executada por somente uma única *thread*, somente uma vez.

Em relação as vantagens fornecidas pela API, estão:

- **Flexibilidade:** A API permite que seja ajustado o número de *threads* em uma região paralela, fornecendo assim flexibilidade para a aplicação.

- **Portabilidade:** A biblioteca OpenMP é distribuída juntamente com o compilador *GNU Compiler Collection* (GCC).

- **Facilidade de Programação:** Com pouco esforço um código sequencial pode ser paralelizado utilizando a API.

## 2.4 TRABALHOS RELACIONADOS

Esta seção apresenta trabalhos relacionados inseridos no contexto da STA. Todos são trabalhos recentes que propõem técnicas para melhorar o desempenho da análise de *timing*, evidenciando a falta de ferramentas eficientes.

OpenTimer (HUANG; WONG, 2015), é uma ferramenta de código aberto que reflete o estado da arte das ferramentas de análise de *timing*. Esta ferramenta realiza a propagação dos atrasos nível por nível como a estratégia de paralelização, a exemplo do que foi implementado no presente trabalho e apresentado na Seção 3.2. Essa propagação dos atrasos é feita utilizando uma estrutura de *pipeline*, dividindo assim o cálculo do atraso em etapas. Essas etapas estão descritas na Seção 3.1.

iTimer C 2.0 (LEE et al., 2015)(primeiro lugar no *TAU Contest 2015*) é uma ferramenta que consegue acelerar o tempo de execução de diversas otimizações incrementais em um circuito utilizando técnicas de *lazy evaluation* e *lazy propagation*. Durante as modificações incrementais no circuito, vértices são marcados como sujos até que uma consulta seja realizada (*lazy evaluation*). Dessa forma, somente serão atualizadas as informações dos vértices que estão na área afetada pelas alterações. Caso informações de atraso geradas por alguma modificação não sejam predominantes, aquele cone lógico afetado não é atualizado, evitando assim atualizações redundantes no cálculo dos atrasos (*lazy propagation*). Embora o trabalho não mencione paralelismo, a aplicação da técnica de nivelamento (seção 3.2) poderia ser aplicada a este trabalho.

Em (HUANG et al., 2016) é apresentado um *framework* para realizar a STA de forma distribuída. O trabalho apresenta o problema real que existe para circuitos complexos onde a STA requer muita memória, surgindo a necessidade de uma alternativa distribuída. A solução proposta é um *framework*, onde partições de um grafo de *timing* são enviadas para clientes que processam a STA e comunicam-se com o servidor.

Em (RAHMAN; TENNAKOON; SECHEN, 2013) são realizadas otimizações do tipo *gate sizing* (KAHNG et al., 2011, p. 221), as quais necessitam de informações sobre o atraso do circuito. Para reduzir o tempo de execução de tais otimizações a STA é realizada utilizando o mecanismo de nivelamento (Seção 3.2) e os atrasos são processados somente quando necessário, evitando atualizações redundantes em uma mesma área do grafo.

# 3 ANÁLISE DE *TIMING* ESTÁTICA

Neste capítulo serão apresentadas as estratégias de paralelismo adotadas por este trabalho e uma análise sobre os algoritmos relacionados.

## 3.1 ALGORITMO ORIGINAL

A análise de *timing* estática segue uma sequência de etapas listadas no Algoritmo 1. Partindo do grafo do circuito com o ordenamento topológico, as interconexões entre os elementos lógicos são modeladas. Então, os atrasos entre os vértices são propagados e os valores de AT, *slew*, RAT e *slack* são calculados usando as Equações 2.1, 2.2, 2.3, 2.4, respectivamente. Finalmente, para cada saída do circuito ou elemento sequencial é realizada uma análise das violações temporais para saber quais violam as restrições de projeto e, se necessário, qual caminho deve ter seu atraso reduzido. A análise das violações é feita utilizando as equações de *setup* e *hold* (2.5, 2.6).

**Algoritmo 1:** Análise de *Timing* Estática

**Entrada**: Grafo de um circuito e seu ordenamento topológico.
**Saída** : Grafo de um circuito com atraso estimado.

1 Modelagem das interconexões
2 Propagações dos atrasos
3 Análise das violações temporais do circuito

Na primeira etapa do Algoritmo 1 a paralelização se torna trivial, pois as interconexões podem ser modeladas de forma independente. Para isso, um laço paralelo pode ser utilizado. Para a segunda etapa outra estratégia é necessária, pois o cálculo para um vértice depende dos valores calculados para seus antecessores. Para extrair o paralelismo nessa etapa, um mecanismo de nivelamento de grafos pode ser utilizado (Seção 3.2), a exemplo do trabalho realizado por (HUANG; WONG, 2015). A terceira etapa também é feita utilizando um laço paralelo, pois os atrasos já foram estimados anteriormente pela segunda etapa.

## 3.2 ALGORITMO DE NIVELAMENTO DO DAG

A estratégia de nivelamento é ilustrada na Figura 12. Para criar a estrutura de dados que representa cada nível do DAG foi utilizado o Algoritmo 2, o qual se beneficia do ordenamento topológico do circuito para desenvolver a separação dos níveis. Ao final do algoritmo, cada nível em $V$ ($V[0]$, $V[1]$, etc.) contém todos os vértices que lhe pertencem, sendo essa informação utilizada para guiar a paralelização da segunda etapa do Algoritmo 1. Cada nível possui um conjunto de vértices independentes entre si, permitindo que sejam processados em paralelo.

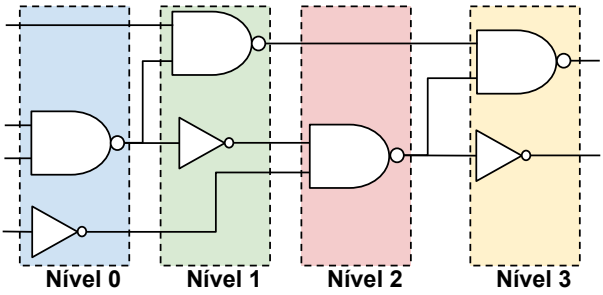



Figura 12 – Exemplo de nivelamento lógico adaptado do DAG.

**Algoritmo 2:** Nivelamento do DAG

**Entrada**: Ordenamento topológico de vértices $T$
**Saída** : Vetor bidimensional de vértices ordenado por níveis $V$

```
1 para t ∈ T faça
2 |   t.nível ← 0
3 |   para i tal que i é antecessor de t faça
4 |   |   t.nível ← max(i.nível + 1, t.nível)
5 |   fim
6 |   V[t.nível] ← t
7 fim
```

O Algoritmo 2 possui uma complexidade linear, sendo esta O(H) onde H é o número de arestas, que sempre será maior do que o número de vértices em um circuito.

## 3.3 DEPENDÊNCIA DE DADOS

Durante a análise de *timing*, para que o atraso de determinado vértice seja calculado, é necessário que todos os antecessores ao vértice já tenham sido calculados. Portanto, a estratégia utilizada é a propagação dos atrasos nível por nível utilizando a estrutura de dados gerada pelo Algoritmo 2, conforme ilustrado pela Figura 13.

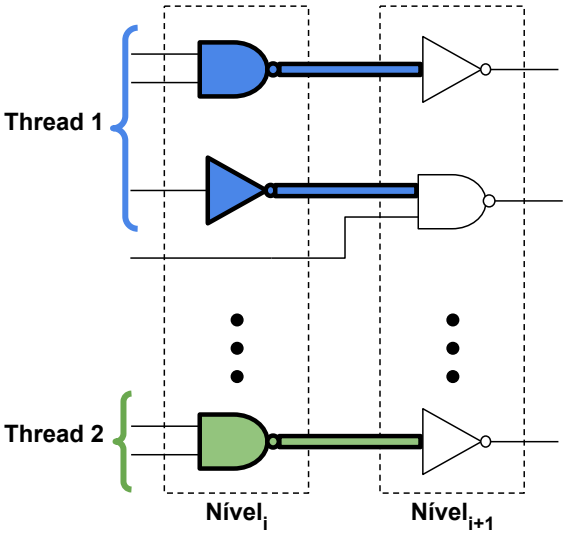



Figura 13 – A propagação dos atrasos em um determinado nível, onde cada uma das *threads* é responsável pelo cálculo em um determinado intervalo.

Uma possível paralelização da segunda etapa do Algoritmo 1 utilizando o mecanismo de nivelamento pode acontecer da seguinte forma:

1. Os vértices de cada nível serão distribuídos entre as *threads*.

2. As *threads* realizam o cálculo do atraso da porta e interconexão ligada na saída da mesma.

3. Dado que uma interconexão possui uma única fonte, isto garante que duas *threads* não vão calcular o atraso de uma mesma interconexão, evitando assim uma condições de corrida.

## 3.4 DISCUSSÃO

Nesta seção serão discutidos detalhes relevantes ao que foi apresentado anteriormente.

Dentre os motivos pelos quais o OpenMP foi escolhido destacam-se: o fato de ser uma API que possui uma linguagem simples e ter sido abordada por algumas disciplinas durante a graduação. A API OpenMP fornece uma interface simples e flexível para a programação, através de diretivas de compilação.

Outro detalhe importante a ser mencionado é que a estrutura dos níveis pode ser reutilizada. Caso alguma modificação seja feita no grafo, basta atualizar apenas os níveis afetados pela modificação. Caso diversas modificações sejam feitas no grafo a atualização dos níveis pode ser postergada de forma a amortizar o custo para manter tal estrutura.

Além da divisão do grafo em níveis, outra estratégia foi abordada utilizando a diretiva *task depend* do OpenMP (ver Figura 18). Esta diretiva basicamente cria uma tarefa onde essa depende de outras tarefas criadas anteriormente. Essa dependência é identificada pelos endereços de memória que são fornecidos como parâmetro para a diretiva.

Apesar de tal estratégia não necessitar da estrutura de níveis, os tempos de execução obtidos com a diretiva *task depend* foram piores do que os obtidos com a versão sequencial original. A principal hipótese para esse resultado está relacionada à granularidade do problema, a qual não parece ser grande o bastante para justificar os sobrecustos de controle e criação de tarefas. Entretanto, isso não foi investigado em maiores detalhes devido às restrições de tempo para a conclusão do presente trabalho.

As diretivas do tipo *task* em OpenMP servem para paralelizar tarefas onde estas podem gerar outras tarefas ou até laços sem limites conhecidos.

# 4 METODOLOGIA E RESULTADOS

Este capítulo apresenta a metodologia adotada juntamente com os diferentes ambientes de testes utilizados. Os resultados estão separados em duas seções: na primeira, o resultado obtido com a máquina do laboratório (máquina local). Na segunda, com uma máquina disponibilizada pelo Grid5000[1].

Os experimentos que serão amostrados foram obtidos utilizando a estrutura de níveis Seção 3.2. Diferentes tipos de escalonamentos foram experimentados, dentre eles o estático foi o que demonstrou ser mais eficiente. Dentre os motivos por ter sido o mais eficiente, destaca-se o fato de que a complexidade do cálculo do atraso de um vértice é a mesma para todos os vértices do grafo. Sendo assim, não é necessário um escalonamento dinâmico para amortizar o desbalanceamento das tarefas, evitando assim o sobrecusto de controle inerente ao escalonamento dinâmico.

## 4.1 AMBIENTES E METODOLOGIA DE TESTES

Experimentos foram desenvolvidos para comparar o desempenho da versão sequencial com a versão paralela proposta, a qual usa *OpenMP*. Foram medidos os tempos referentes às etapas citadas no Algoritmo 2.

Foi utilizada a infraestrutura fornecida pela competição ICCAD 2015 CAD *Contest* (Problema C: posicionamento guiado por atraso) que adota arquivos com os formatos utilizados pela indústria de circuitos integrados (HUANG; WONG, 2015). Os resultados foram obtidos com um mínimo de 20 execuções para cada circuito, informações sobre os circuitos são apresentadas na Tabela 1.

| | sb18 | sb4 | sb16 | sb5 | sb1 | sb3 | sb10 | sb7 |
|---|---|---|---|---|---|---|---|---|
| #Células | 768068 | 795645 | 981559 | 1086888 | 1209716 | 1213253 | 1876103 | 1931639 |
| #Interconexões | 771542 | 802513 | 999902 | 1100825 | 1215710 | 1224979 | 1898119 | 1933945 |

Tabela 1 – Configurações dos circuitos utilizados nos testes.

O ambiente de testes inclui duas máquinas, uma delas fornecida pelo laboratório ECL e a outra disponibilizada disponibilizada pelo

[1] Ambiente de *grid* computacional francês (GRID5000, 2016).

Grid5000 cujas configurações respectivas são detalhadas na Tabela 2.

| | Processador | Número de Núcleos | Memória | Sistema Operacional | Arquitetura |
|---|---|---|---|---|---|
| #Máquina Local | Intel® Core™ i5-4460 CPU 3.20GHz | **4** | 32GB | Debian 8 Jessie | UMA |
| #Máquina Grid5000 | **4**× AMD Opteron 6174 de 12 núcleos | $\mathbf{12 \times 4 = 48}$ | 256GB | Debian Squeeze | NUMA |

Tabela 2 – Configurações dos ambientes de teste.

Foi implementada uma versão paralela para a ferramenta de STA, que utiliza a biblioteca de grafos *Lemon graph library* v1.3.1. O projeto foi compilado com *gcc* versão 5.4.1. Para fazer uso da memória *cache* de maneira eficiente, foi utilizada a biblioteca *Portable Hardware Locality* (hwloc) v1.11.2, onde as *threads* foram mapeadas aos núcleos.

### 4.1.1 Métricas Básicas de Desempenho

**SpeedUp:** É calculado pela Equação 4.1, indica quantas vezes o programa é mais rápido que a versão sequencial. É um valor adimensional obtido pela razão entre a o tempo da versão sequencial com o tempo da versão paralela.

Caso o *SpeedUp* calculado seja $> 1$ significa que a versão paralela reduziu o tempo de execução, caso o valor seja $< 1$ significa que a versão paralela obteve um desempenho inferior a versão sequencial.

$$SpeedUp = \frac{T_{seq.}}{T_{par.}} \tag{4.1}$$

**Eficiência:** É calculado pela Equação 4.2, indica qual a taxa de utilização média das unidades utilizadas no paralelismo. Mostra o grau de aproveitamento das unidades utilizadas.

$$Eficiência = \frac{SpeedUp}{N_{unidades}} \tag{4.2}$$

Observações importantes:

- O ideal seria que a eficiência fique próxima de 100%, porém a dependência de dados pode fazer unidades de processamento aguardarem por dados produzidos por outras unidades, reduzindo assim a eficiência.

- Outra observação importante é que não necessariamente a maior eficiência é obtida com o maior número de *threads*. Pode ser

que aumentar o número de *threads* não somente reduza o tempo de execução como também a eficiência, pois pode aumentar o sobrecusto de sincronização das unidades ativas.

## 4.2 RESULTADOS COM A MÁQUINA LOCAL

A Figura 14 ilustra os resultados obtidos com a técnica de processamento dos níveis do grafo em paralelo. Os tempos de execução incluem a criação da estrutura dos níveis apresentada na Seção 3.2.

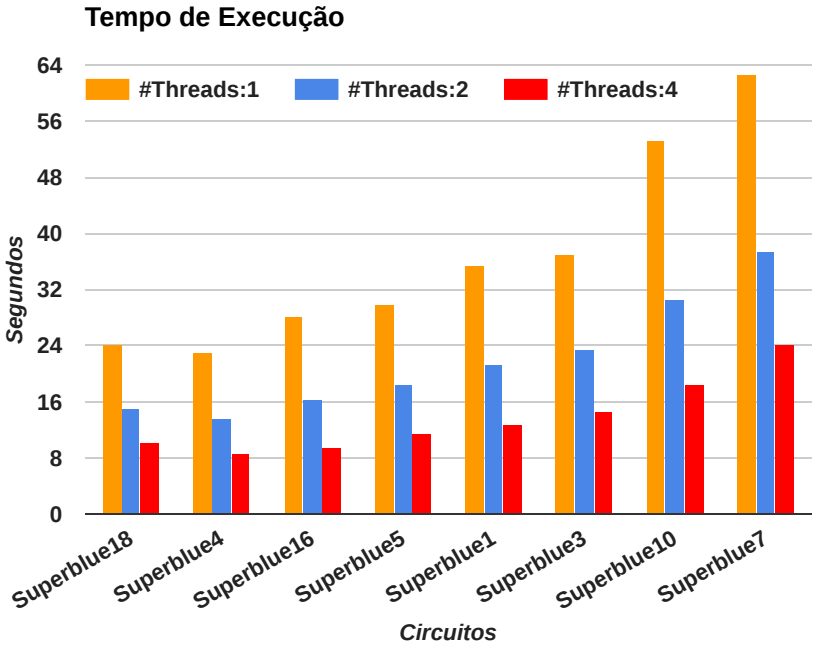



Figura 14 – Tempos de execução variando de uma a quatro *threads* na Máquina Local.

Para o circuito Superblue4 é possível notar que, embora ele possua um maior número de células em relação ao primeiro Superblue18, o tempo de execução foi menor evidenciando que a complexidade das conexões entre os elementos também deve ser levada em consideração.

Ainda na Figura 14, é possível observar que com o aumento do número de *threads* o tempo de execução foi reduzido.

A Figura 15 ilustra o *speedup* normalizado em relação a versão sequencial.

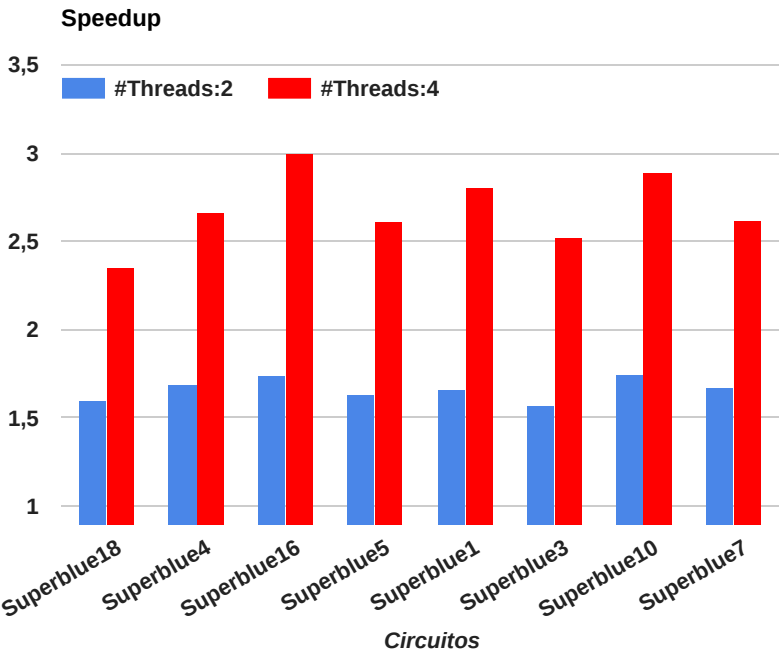



Figura 15 – *Speedup* normalizado obtido para os diferentes circuitos.

Também é possível observar que, de todos os circuitos, o Superblue16 foi o que conseguiu o melhor *speedup* para quatro *threads*, o que é um bom indício de que este possui um comportamento mais regular em relação à estrutura topológica de níveis.

Em média, a paralelização leva a um *speedup* de $1,66$ para duas *threads* e $2,68$ para quatro *threads*. Portanto, a maior eficiência 83% é obtida com duas *threads*.

## 4.3 RESULTADOS COM A MÁQUINA GRID

Esta seção apresenta os resultados obtidos em uma máquina NUMA *quad-socket* com total de 48 núcleos. O objetivo desses testes é verificar a escalabilidade da técnica quando aplicada em um ambiente com maior suporte de multiprocessamento.

Para cada circuito, os tempos de execução são apresentados para evidenciar a escalabilidade da técnica em relação ao aumento do número de *threads*. Os tempos de execução incluem a criação da estrutura dos níveis apresentada na Seção 3.2.

A Figura 16, que está em escala logarítmica, mostra os tempos médios de execução para a Máquina Grid.

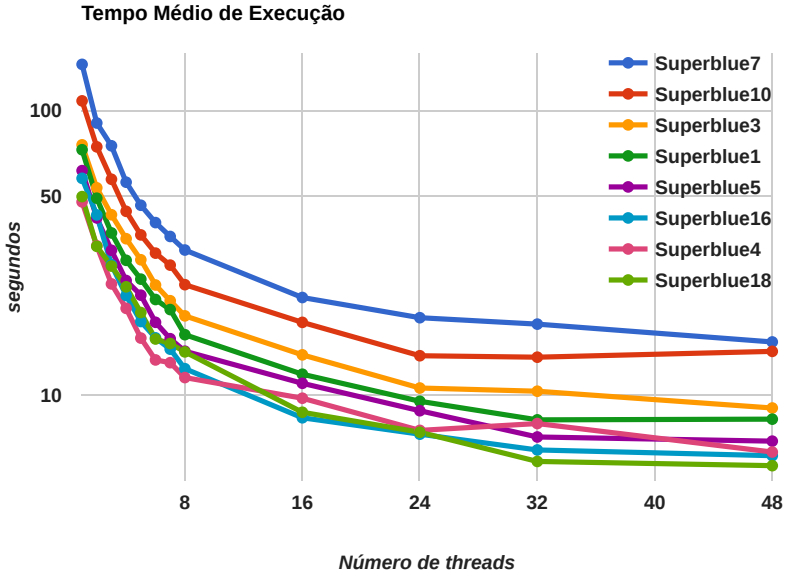



Figura 16 – Tempo médio de execução em escala logarítmica, obtido com a Máquina Grid.

Na Figura 16, é possível observar que com o aumento do número de *threads* o tempo de execução foi reduzido em uma ordem de grandeza para todos os circuitos.

Para o circuito Superblue10, por exemplo, é possível notar que ocorreu um pequeno aumento no tempo de execução de 32 *threads* para 48 *threads*. Portanto, como já mencionado anteriormente, aumentar o número de *threads* pode não reduzir o tempo de execução, podendo inclusive aumentá-lo.

A figura 17 mostra o *speedup* médio e ideal alcançado com a Máquina Grid.

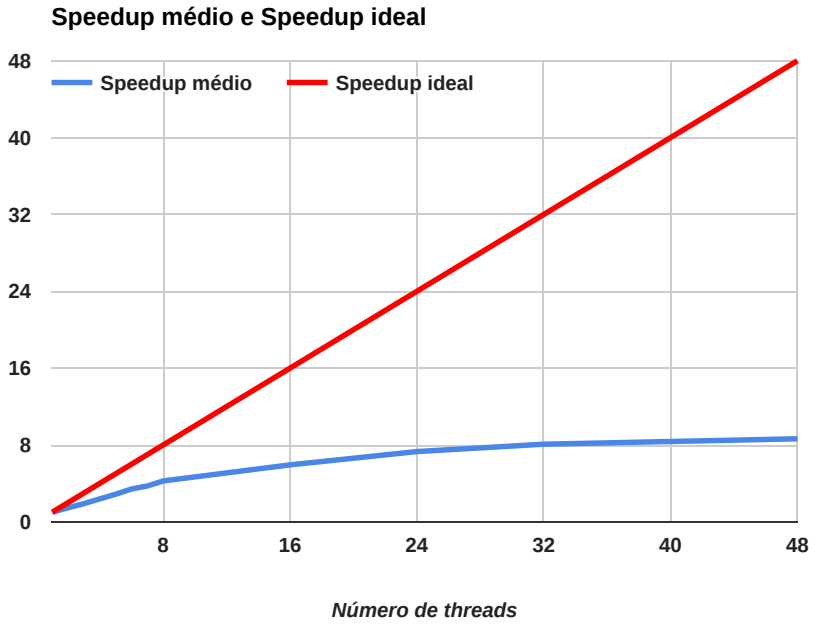



Figura 17 – *Speedup* médio e ideal em relação ao aumento no número de *threads*.

Embora o tempo de execução seja reduzido com o aumento do número de *threads*, a eficiência da técnica mostra-se muito distante da ideal como pode ser observado na Figura 17.

A eficiência da técnica se mantém próxima de 50% até oito *threads*, após isto cai drasticamente pelo fato de que a máquina possui quatro processadores de doze núcleos. Sendo assim, o processamento fica todo concentrado em um dos núcleos e portanto, o sobrecusto de comunicação é menor. Conforme o número de *threads* aumenta (maior que doze) o processamento se distribui entre os processadores aumentando também o sobrecusto de comunicação.

Portanto, existe, para cada aplicação um número ideal de *threads* que leva a obter um melhor desempenho, não sendo verdade que quanto maior o número de *threads* melhor será o desempenho (NAVAUX; ROSE; PILLA, 2011, p. 28).

Vale ressaltar que na prática as arquiteturas computacionais usualmente utilizadas não necessitam de tanto poder computacional, pois a STA é por definição uma estimativa simples, caso necessário métodos complexos são utilizados com suporte de *clusters* computacionais.

# 5 CONCLUSÕES

Este trabalho investigou o impacto da paralelização do algoritmo de STA através da divisão do circuito em níveis lógicos, para avaliar a técnica foram utilizados os circuitos fornecidos pela competição ICCAD 2015.

Os resultados com a Máquina Local demonstraram um *speedup* médio para os circuitos de aproximadamente $2,68$ para 4 *threads*, desconsiderando a criação da estrutura de dados.

Experimentos foram realizados em uma máquina com mais núcleos para verificar o comportamento da técnica, demonstrando uma redução na ordem de grandeza dos tempos de execução para todos os circuitos.

De fato, o paralelismo na análise de *timing* pode ser explorado como demonstrado pelos resultados. Porém, em uma arquitetura (NUMA) o tempo de acesso à memória pode reduzir consideravelmente o desempenho obtido pelo paralelismo, como foi observado na Figura 17. Essa redução considerável no desempenho ocorre quando o número de *threads* maior que o número de núcleos em um processador, fazendo com que tarefas se distribuam entre os processadores, ocorrendo assim comunicação inter-processador.

## 5.1 TRABALHOS FUTUROS

Para extrair melhor o proveito do paralelismo da análise de *timing*, trabalhos futuros incluem:

- Implementar a STA de forma distribuída, para explorar melhor o paralelismo e reduzir a necessidade de uma máquina que possua uma grande quantidade de memória.
- Implementar diferentes etapas da STA utilizando *pipeline* para explorar mais paralelismo.
- Adotar outros tipos de estrutura de dados para melhorar o desempenho da memória *cache*.
- Verificar a possibilidade o uso de um *framework* especializado para processamento paralelo de um DAG como por exemplo *GAS - Gather Apply Scatter*[1].

[1] GAS é uma abordagem na área de *Big Data* para ranqueamento de páginas.

# REFERÊNCIAS

ECLUFSC. *Ophidian*. GitHub, 2016. <https://github.com/sheiny/ophidian/tree/TCC>.

GRID5000. *Ambiente de grid computacional francês*. 2016. Acesso em: 29-08-2016. <http://digitalis.inria.fr/index.php/Idfreeze>.

HU, J.; SCHAEFFER, G.; GARG, V. Tau 2015 contest on incremental timing analysis: Incremental timing and cppr analysis. In: IEEE PRESS. *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. [S.l.], 2015. p. 882–889.

HUANG, T.-W.; WONG, M. D. Opentimer: A high-performance timing analysis tool. In: IEEE PRESS. *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. [S.l.], 2015. p. 895–902.

HUANG, T.-W. et al. A distributed timing analysis framework for large designs. In: ACM. *Proceedings of the 53rd Annual Design Automation Conference*. [S.l.], 2016. p. 116.

KAHNG, A. B. et al. *VLSI Physical Design: From Graph Partitioning to Timing Closure*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2011. ISBN 9789048195909.

LEE, P.-Y. et al. itimerc 2.0: Fast incremental timing and cppr analysis. In: IEEE. *Computer-Aided Design (ICCAD), 2015 IEEE/ACM International Conference on*. [S.l.], 2015. p. 890–894.

NAVAUX, P. O.; ROSE, C. A. D.; PILLA, L. L. Fundamentos das arquiteturas para processamento paralelo e distribuído. *XI Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul-2011-Porto Alegre, RS*, p. 22–59, 2011.

OPENMP. *OpenMP Application Program Interface Version 4.5*. 2015. <http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf>.

RAHMAN, M.; TENNAKOON, H.; SECHEN, C. Library-based cell-size selection using extended logical effort. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, IEEE, v. 32, n. 7, p. 1086–1099, 2013.

SAPATNEKAR, S. *Timing*. [S.l.]: Springer Science & Business Media, 2004.

VISWESWARIAH, C. Within die variations in timing: From derating to cppr to statistical methods. *ICCAD Tutorial. IBM, Inc*, 2007.

WESTE, N.; HARRIS, D. *CMOS VLSI Design: A Circuits and Systems Perspective*. 4th. ed. USA: Addison-Wesley Publishing Company, 2010. ISBN 0321547748, 9780321547743.

# ANEXO A – Resultados com *OpenMP tasks*

Experimentos com a diretiva *task depend* em OpenMP foram realizados em uma versão anterior da ferramenta desenvolvida no laboratório ECL. A Figura 18, ilustra os resultados obtidos com a diretiva *task depend*. A granularidade das tarefas, neste experimento, equivale a um vértice no grafo de *timing*, o que pode ter contribuído para piorar o resultado da solução.

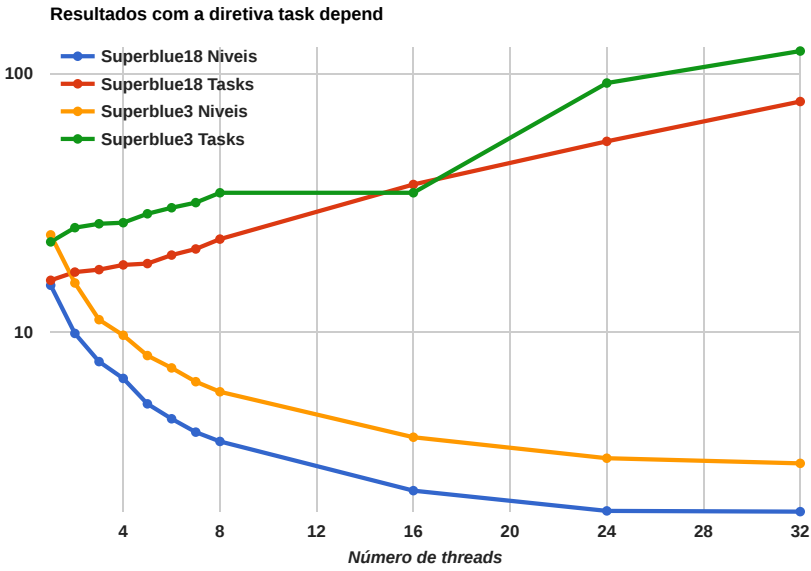



Figura 18 – Tempos de execução utilizando a diretiva em OpenMP *task depend* para realizar a STA.

# ANEXO B – Artigo sobre o TCC

# PARALELISMO EM ANÁLISE DE *TIMING* ESTÁTICA

**Sheiny Fabre Almeida**[1], **Laércio Lima Pilla**[1], **José Luís Almada Güntzel**[1]

[1]Instituto de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Brazil

{laercio.pilla,j.guntzel}@ufsc.br,sheiny.fabre@gmail.com

***Abstract.*** *This work seeks to study the use of parallel programming applied to Static Timing Analysis (STA), which is a technique used to estimate the expected delay of a circuit during the physical synthesis step. The study involves the identification of parallel opportunities and possible improvements in the data structure in order to improve the parallel solution· Lastly, this work includes the development of parallel solutions based on toolkit developed in Embedded Computing Laboratory (ECL) from Federal University of Santa Catarina (UFSC).*

***Resumo.*** *O presente trabalho visa estudar o uso de programação paralela aplicada à Análise de Timing Estática - Static Timing Analysis (STA), uma técnica utilizada para estimar o atraso de um circuito durante a etapa de síntese física. O estudo envolve a identificação de oportunidades de paralelismo e possíveis melhorias na estrutura de dados para aprimorar a solução paralela. Por fim, o trabalho inclui o desenvolvimento de soluções paralelas com base no ferramental desenvolvido no Laboratório de Computação Embarcada (ECL) da Universidade Federal de Santa Catarina (UFSC).*

## 1. Introdução

Os Circuitos Integrados - *Integrated Circuits* (CIs) constituem o núcleo de qualquer equipamento eletrônico contemporâneo. O projeto de CIs requer uma sequência de etapas, dentre as quais a síntese física é a responsável por gerar a descrição fabricável. A síntese física busca posicionar os *layouts* das portas lógicas e *flip-flops* (referenciados por "células") sobre uma região 2-D, realizando as conexões necessárias entre estes e entre os *flip-flops* e o sinal de relógio [Kahng et al. 2011, p. 8].

O posicionamento de células resultará em uma configuração topológica com um determinado atraso. A fim de satisfazer as restrições de atraso do circuito para atingir a frequência de relógio alvo, diversas técnicas de otimização são aplicadas na síntese física. Dentre tais técnicas citam-se *gate sizing*, inserção de *buffers* e Posicionamento Incremental Guiado por Atraso - *Incremental Timing Driven Placement* (ITDP) [Kahng et al. 2011, p. 221].

Todas essas técnicas necessitam de estimativas precisas de atraso do circuito, a fim de guiar a otimização e garantir a convergência das técnicas empregadas. Além disso, tal estimativa deve ser obtida com o menor tempo de execução possível, pois ela precisa ser feita a cada iteração da otimização. A STA é utilizada para tal propósito. Por questões de eficiência computacional, a STA é um método utilizado para computar o atraso esperado de um circuito digital sem realizar qualquer tipo de simulação [Kahng et al. 2011, p. 222].

Dada a complexidade crescente dos projetos de CIs, a verificação temporal vem se tornando rapidamente uma forte limitadora para a síntese física [Huang et al. 2016]. O algoritmo mais conhecido e trivial para STA é a análise de *timing* topológica, onde o circuito é representado como um Grafo Acíclico Direcionado - *Direct Acyclic Graph* (DAG) e é percorrido em ordem topológica, propagando os atrasos dos elementos de maneira cumulativa, das entradas para as saídas do circuito [Hu et al. 2015]. **Tendo em vista que a análise de *timing* topológica não apresenta paralelismo em sua forma padrão, este trabalho propõe um estudo sobre técnicas de paralelismo que possam ser empregadas na análise de *timing* estática.**

## 2. Análise de *Timing* Estática

A análise de *timing* estática segue uma sequência de etapas listadas no Algoritmo 1. Partindo do grafo do circuito com o ordenamento topológico, as interconexões entre os elementos lógicos são modeladas. Então, os atrasos entre os vértices são propagados. Finalmente, para cada saída do circuito ou elemento sequencial é realizada uma análise das violações temporais.

**Algoritmo 1:** Análise de *Timing* Estática

**Entrada:** Grafo de um circuito e seu ordenamento topológico.
**Saída :** Grafo de um circuito com atraso estimado.
1 Modelagem das interconexões
2 Propagações dos atrasos
3 Análise das violações temporais do circuito

Na primeira etapa do Algoritmo 1 a paralelização se torna trivial, pois as interconexões podem ser modeladas de forma independente. Para isso, um laço paralelo pode ser utilizado. Para a segunda etapa outra estratégia é necessária, pois o cálculo para um vértice depende dos valores calculados para seus antecessores. Para extrair o paralelismo nessa etapa, um mecanismo de nivelamento de grafos pode ser utilizado (Seção 3), a exemplo do trabalho realizado por [Huang and Wong 2015]. A terceira etapa também é feita utilizando um laço paralelo, pois os atrasos já foram estimados anteriormente pela segunda etapa.

## 3. Algoritmo de Nivelamento do DAG

A estratégia de nivelamento é ilustrada na Figura 1. Para criar a estrutura de dados que representa cada nível do DAG foi utilizado o Algoritmo 2, o qual se beneficia do ordenamento topológico do circuito para desenvolver a separação dos níveis. Ao final do algoritmo, cada nível em $V$ ($V[0]$, $V[1]$, etc.) contém todos os vértices que lhe pertencem, sendo essa informação utilizada para guiar a paralelização da segunda etapa do Algoritmo 1. Cada nível possui um conjunto de vértices independentes entre si, permitindo que sejam processados em paralelo.

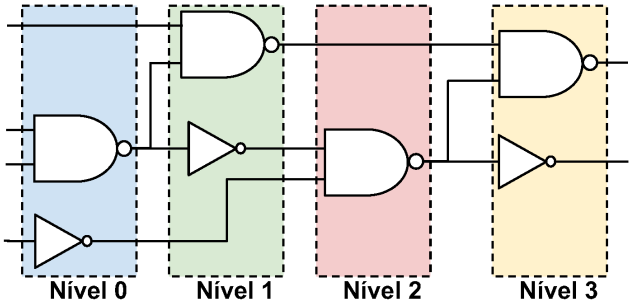



**Figura 1. Exemplo de nivelamento lógico adaptado do DAG.**

**Algoritmo 2:** Nivelamento do DAG

**Entrada:** Ordenamento topológico de vértices $T$
**Saída :** Vetor bidimensional de vértices ordenado por níveis $V$

```
1 para t ∈ T faça
2     t.nível ← 0
3     para i tal que i é antecessor de t faça
4         t.nível ← max(i.nível + 1, t.nível)
5     fim
6     V[t.nível] ← t
7 fim
```

O Algoritmo 2 possui uma complexidade linear, sendo esta O(H) onde H é o número de arestas, que sempre será maior do que o número de vértices em um circuito.

Metodologia e Resultados Este capítulo apresenta a metodologia adotada juntamente com os diferentes ambientes de testes utilizados. Os resultados estão separados em duas seções: na primeira, o resultado obtido com a máquina do laboratório (máquina local). Na segunda, com uma máquina disponibilizada pelo Grid5000[1].

Os experimentos que serão amostrados foram obtidos utilizando a estrutura de níveis Seção 3. Diferentes tipos de escalonamentos foram experimentados, dentre eles o estático foi o que demonstrou ser mais eficiente. Dentre os motivos por ter sido o mais eficiente, destaca-se o fato de que a complexidade do cálculo do atraso de um vértice é a mesma para todos os vértices do grafo. Sendo assim, não é necessário um escalonamento dinâmico para amortizar o desbalanceamento das tarefas, evitando assim o sobrecusto de controle inerente ao escalonamento dinâmico.

## 4. Ambientes e Metodologia de Testes

Experimentos foram desenvolvidos para comparar o desempenho da versão sequencial com a versão paralela proposta, a qual usa *OpenMP*. Foram medidos os tempos referentes às etapas citadas no Algoritmo 2.

[1] Ambiente de *grid* computacional francês [grid5000 2016].

Foi utilizada a infraestrutura fornecida pela competição ICCAD 2015 CAD *Contest* (Problema C: posicionamento guiado por atraso) que adota arquivos com os formatos utilizados pela indústria de circuitos integrados [Huang and Wong 2015]. Os resultados foram obtidos com um mínimo de 20 execuções para cada circuito, informações sobre os circuitos são apresentadas na Tabela 1.

| | sb18 | sb4 | sb16 | sb5 | sb1 | sb3 | sb10 | sb7 |
|---|---|---|---|---|---|---|---|---|
| #Células | 768068 | 795645 | 981559 | 1086888 | 1209716 | 1213253 | 1876103 | 1931639 |
| #Interconexões | 771542 | 802513 | 999902 | 1100825 | 1215710 | 1224979 | 1898119 | 1933945 |

**Tabela 1. Configurações dos circuitos utilizados nos testes.**

O ambiente de testes inclui duas máquinas, uma delas fornecida pelo laboratório ECL e a outra disponibilizada disponibilizada pelo Grid5000 cujas configurações respectivas são detalhadas na Tabela 2.

| | Processador | Número de Núcleos | Memória | Sistema Operacional | Arquitetura |
|---|---|---|---|---|---|
| #Máquina Local | Intel® Core™ i5-4460 CPU 3.20GHz | **4** | 32GB | Debian 8 Jessie | UMA |
| #Máquina Grid5000 | 4× AMD Opteron 6174 de 12 núcleos | $12 \times 4 = 48$ | 256GB | Debian Squeeze | NUMA |

**Tabela 2. Configurações dos ambientes de teste.**

Foi implementada uma versão paralela para a ferramenta de STA, que utiliza a biblioteca de grafos *Lemon graph library* v1.3.1. O projeto foi compilado com *gcc* versão 5.4.1. Para fazer uso da memória *cache* de maneira eficiente, foi utilizada a biblioteca *Portable Hardware Locality* (hwloc) v1.11.2, onde as *threads* foram mapeadas aos núcleos.

### 4.1. Métricas Básicas de Desempenho

**SpeedUp:** É calculado pela Equação 1, indica quantas vezes o programa é mais rápido que a versão sequencial. É um valor adimensional obtido pela razão entre a o tempo da versão sequencial com o tempo da versão paralela.

Caso o *SpeedUp* calculado seja $> 1$ significa que a versão paralela reduziu o tempo de execução, caso o valor seja $< 1$ significa que a versão paralela obteve um desempenho inferior a versão sequencial.

$$SpeedUp = \frac{T_{seq.}}{T_{par.}} \tag{1}$$

**Eficiência:** É calculado pela Equação 2, indica qual a taxa de utilização média das unidades utilizadas no paralelismo. Mostra o grau de aproveitamento das unidades utilizadas.

$$Eficiência = \frac{SpeedUp}{N_{unidades}} \tag{2}$$

Observações importantes:

- O ideal seria que a eficiência fique próxima de $100\%$, porém a dependência de dados pode fazer unidades de processamento aguardarem por dados produzidos por outras unidades, reduzindo assim a eficiência.
- Outra observação importante é que não necessariamente a maior eficiência é obtida com o maior número de *threads*. Pode ser que aumentar o número de *threads* não somente reduza o tempo de execução como também a eficiência, pois pode aumentar o sobrecusto de sincronização das unidades ativas.

## 5. Resultados com a Máquina Local

A Figura 2 ilustra os resultados obtidos com a técnica de processamento dos níveis do grafo em paralelo. Os tempos de execução incluem a criação da estrutura dos níveis apresentada na Seção 3.

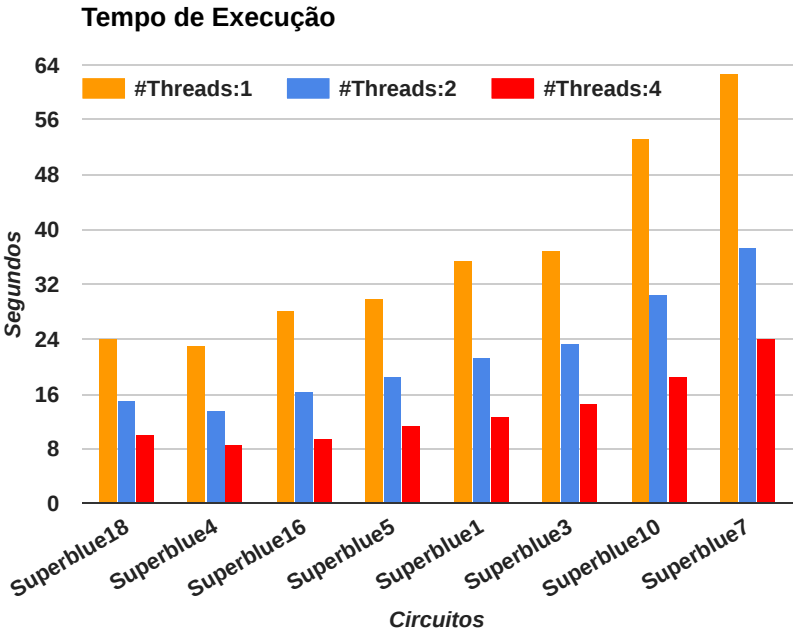



**Figura 2. Tempos de execução variando de uma a quatro *threads* na Máquina Local.**

Para o circuito Superblue4 é possível notar que, embora ele possua um maior número de células em relação ao primeiro Superblue18, o tempo de execução foi menor evidenciando que a complexidade das conexões entre os elementos também deve ser levada em consideração.

Ainda na Figura 2, é possível observar que com o aumento do número de *threads* o tempo de execução foi reduzido.

A Figura 3 ilustra o *speedup* normalizado em relação a versão sequencial.

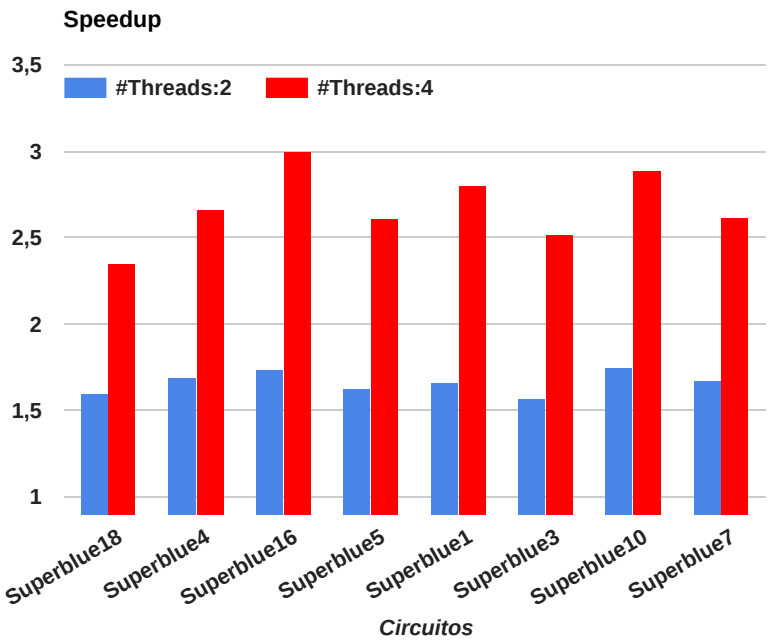



**Figura 3. *Speedup* normalizado obtido para os diferentes circuitos.**

Também é possível observar que, de todos os circuitos, o Superblue16 foi o que conseguiu o melhor *speedup* para quatro *threads*, o que é um bom indício de que este possui um comportamento mais regular em relação à estrutura topológica de níveis.

Em média, a paralelização leva a um *speedup* de $1,66$ para duas *threads* e $2,68$ para quatro *threads*. Portanto, a maior eficiência $83\%$ é obtida com duas *threads*.

## 6. Resultados com a Máquina Grid

Esta seção apresenta os resultados obtidos em uma máquina NUMA *quad-socket* com total de 48 núcleos. O objetivo desses testes é verificar a escalabilidade da técnica quando aplicada em um ambiente com maior suporte de multiprocessamento.

Para cada circuito, os tempos de execução são apresentados para evidenciar a escalabilidade da técnica em relação ao aumento do número de *threads*. Os tempos de execução incluem a criação da estrutura dos níveis apresentada na Seção 3.

A Figura 4, que está em escala logarítmica, mostra os tempos médios de execução para a Máquina Grid.

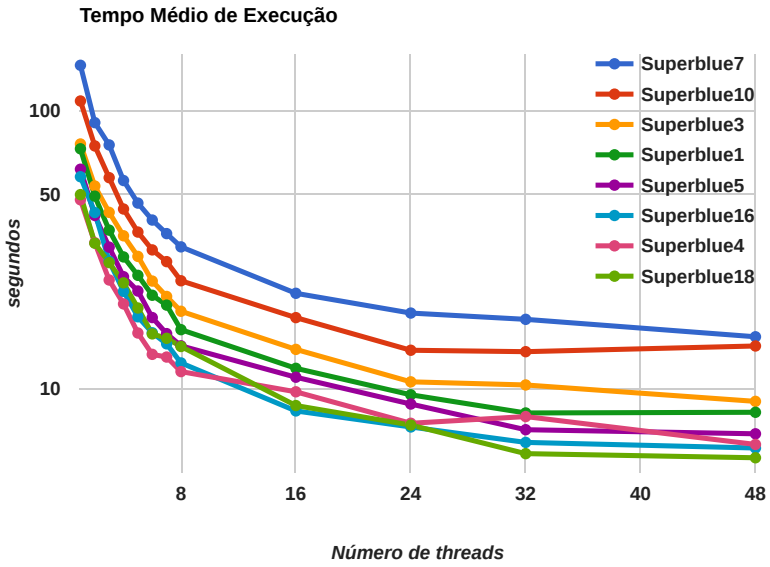



**Figura 4. Tempo médio de execução em escala logarítmica, obtido com a Máquina Grid.**

Na Figura 4, é possível observar que com o aumento do número de *threads* o tempo de execução foi reduzido em uma ordem de grandeza para todos os circuitos.

Para o circuito Superblue10, por exemplo, é possível notar que ocorreu um pequeno aumento no tempo de execução de $32$ *threads* para $48$ *threads*. Portanto, como já mencionado anteriormente, aumentar o número de *threads* pode não reduzir o tempo de execução, podendo inclusive aumentá-lo.

A figura 5 mostra o *speedup* médio e ideal alcançado com a Máquina Grid.

Embora o tempo de execução seja reduzido com o aumento do número de *threads*, a eficiência da técnica mostra-se muito distante da ideal como pode ser observado na Figura 5.

A eficiência da técnica se mantém próxima de $50\%$ até oito *threads*, após isto cai drasticamente pelo fato de que a máquina possui quatro processadores de doze núcleos. Sendo assim, o processamento fica todo concentrado em um dos núcleos e portanto, o sobrecusto de comunicação é menor. Conforme o número de *threads* aumenta (maior que doze) o processamento se distribui entre os processadores aumentando também o sobrecusto de comunicação.

Portanto, existe, para cada aplicação um número ideal de *threads* que leva a obter um melhor desempenho, não sendo verdade que quanto maior o número de *threads* melhor será o desempenho [Navaux et al. 2011, p. 28].

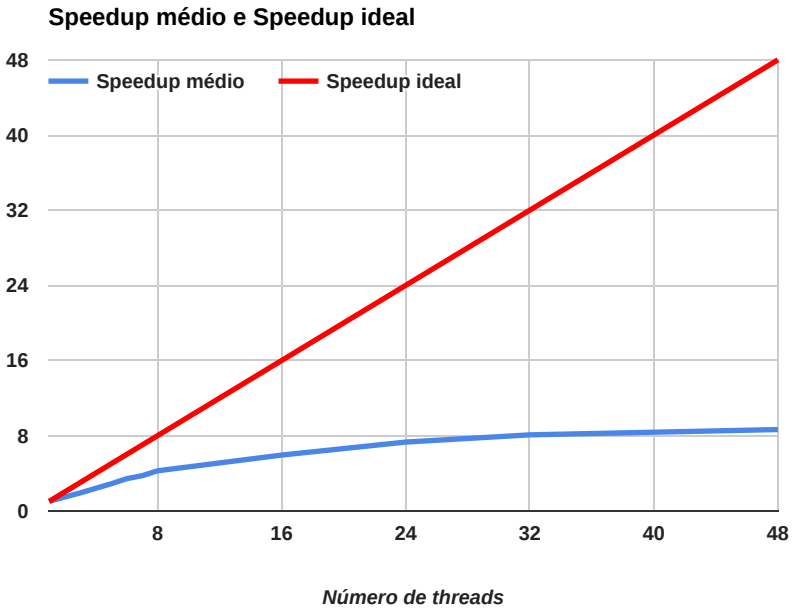



**Figura 5. *Speedup* médio e ideal em relação ao aumento no número de *threads*.**

Vale ressaltar que na prática as arquiteturas computacionais usualmente utilizadas não necessitam de tanto poder computacional, pois a STA é por definição uma estimativa simples, caso necessário métodos complexos são utilizados com suporte de *clusters* computacionais.

## 7. Conclusões

Este trabalho investigou o impacto da paralelização do algoritmo de STA através da divisão do circuito em níveis lógicos, para avaliar a técnica foram utilizados os circuitos fornecidos pela competição ICCAD 2015.

Os resultados com a Máquina Local demonstraram um *speedup* médio para os circuitos de aproximadamente $2,68$ para 4 *threads*, desconsiderando a criação da estrutura de dados.

Experimentos foram realizados em uma máquina com mais núcleos para verificar o comportamento da técnica, demonstrando uma redução na ordem de grandeza dos tempos de execução para todos os circuitos.

De fato, o paralelismo na análise de *timing* pode ser explorado como demonstrado pelos resultados. Porém, em uma arquitetura (NUMA) o tempo de acesso à memória pode reduzir consideravelmente o desempenho obtido pelo paralelismo, como foi observado na Figura 5. Essa redução considerável no desempenho ocorre quando o número de *threads*

maior que o número de núcleos em um processador, fazendo com que tarefas se distribuam entre os processadores, ocorrendo assim comunicação inter-processador.

## 8. Trabalhos Futuros

Para extrair melhor o proveito do paralelismo da análise de *timing*, trabalhos futuros incluem:

- Implementar a STA de forma distribuída, para explorar melhor o paralelismo e reduzir a necessidade de uma máquina que possua uma grande quantidade de memória.
- Implementar diferentes etapas da STA utilizando *pipeline* para explorar mais paralelismo.
- Adotar outros tipos de estrutura de dados para melhorar o desempenho da memória *cache*.
- Verificar a possibilidade o uso de um *framework* especializado para processamento paralelo de um DAG como por exemplo *GAS - Gather Apply Scatter*[2].

## Referências


grid5000 (2016). Ambiente de grid computacional francês. Acesso em: 29-08-2016.

Hu, J., Schaeffer, G., and Garg, V. (2015). Tau 2015 contest on incremental timing analysis: Incremental timing and cppr analysis. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 882–889. IEEE Press.

Huang, T.-W. and Wong, M. D. (2015). Opentimer: A high-performance timing analysis tool. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 895–902. IEEE Press.

Huang, T.-W., Wong, M. D., Sinha, D., Kalafala, K., and Vankateswaran, N. (2016). A distributed timing analysis framework for large designs. In *Proceedings of the 53rd Annual Design Automation Conference*, page 116. ACM.

Kahng, A. B., Lienig, J., Markov, I. L., and Hu, J. (2011). *VLSI Physical Design: From Graph Partitioning to Timing Closure*. Springer Publishing Company, Incorporated, 1st edition.

Navaux, P. O., De Rose, C. A., and Pilla, L. L. (2011). Fundamentos das arquiteturas para processamento paralelo e distribuído. *XI Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul-2011-Porto Alegre, RS*, pages 22–59.


[2]GAS é uma abordagem na área de *Big Data* para ranqueamento de páginas.

# ANEXO C – Código fonte da ferramenta em C++

```
/*
 * Copyright 2016 Ophidian
Licensed to the Apache Software Foundation (ASF) under one
or more contributor license agreements.  See the NOTICE file
distributed with this work for additional information
regarding copyright ownership.  The ASF licenses this file
to you under the Apache License, Version 2.0 (the
"License"); you may not use this file except in compliance
with the License.  You may obtain a copy of the License at

  http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing,
software distributed under the License is distributed on an
"AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
KIND, either express or implied.  See the License for the
specific language governing permissions and limitations
under the License.
 */

#include "../timing/library.h"
#include "../timing/graph.h"
#include "../netlist/netlist.h"
#include "../interconnection/rc_tree.h"

#include <lemon/connectivity.h>

#include <boost/units/limits.hpp>
#include <boost/units/cmath.hpp>
#include <functional>

#include "ceff.h"
#include "design_constraints.h"

#include "graph_nodes_timing.h"
#include "graph_arcs_timing.h"

#include <omp.h>
#include <lemon/path.h>

#ifndef OPHIDIAN_TIMING_GENERIC_STA_H
#define OPHIDIAN_TIMING_GENERIC_STA_H

namespace ophidian {
namespace timing {

struct optimistic {
    using TimingType = boost::units::quantity< boost::units
        ::si::time >;

```

```
50      TimingType operator()(const TimingType &a, const
            TimingType &b) const {
51          return std::min(a, b);
52      }
53      TimingType inverted(const TimingType &a, const
            TimingType &b) const {
54          return std::max(a, b);
55      }
56      static double slack_signal() {
57          return -1.0;
58      }
59      static TimingType best() {
60          return std::numeric_limits<TimingType >::infinity();
61      }
62      static TimingType worst() {
63          return -std::numeric_limits<TimingType >::infinity()
                ;
64      }
65  };
66
67
68  struct pessimistic {
69      using TimingType = boost::units::quantity< boost::units
            ::si::time >;
70
71      TimingType operator()(const TimingType &a, const
            TimingType &b) const {
72          return std::max(a, b);
73      }
74      TimingType inverted(const TimingType &a, const
            TimingType &b) const {
75          return std::min(a, b);
76      }
77      static double slack_signal() {
78          return 1.0;
79      }
80      static TimingType best() {
81          return -std::numeric_limits<TimingType >::infinity()
                ;
82      }
83      static TimingType worst() {
84          return std::numeric_limits<TimingType >::infinity();
85      }
86  };
87
88
89  struct timing_data {
90      const library & lib;
91      graph_nodes_timing nodes;
92      graph_arcs_timing arcs;
93
94
```

```
95      timing_data(const library & lib, const graph& g):
96          lib(lib),
97          nodes(g.G()),
98          arcs(g.G())
99      {
100
101     }
102
103 };
104
105
106 struct graph_and_topology {
107     const graph & g;
108     const netlist::netlist & netlist;
109     std::vector<lemon::ListDigraph::Node> sorted;
110     std::vector< std::vector<lemon::ListDigraph::Node> >
            levels;
111     std::vector<lemon::ListDigraph::Node> sorted_drivers;
112     graph_and_topology(const graph & G, const netlist::
            netlist & netlist, const library & lib);
113
114 };
115
116
117 struct test_calculator {
118
119     const graph_and_topology & topology;
120     timing_data & early;
121     timing_data & late;
122     boost::units::quantity< boost::units::si::time >
            clock_period;
123
124     void compute_tests();
125
126
127 };
128
129 template <class WireDelayModel, class MergeStrategy>
130 class generic_sta
131 {
132     using SlewType = boost::units::quantity< boost::units::
            si::time >;
133     using CapacitanceType = boost::units::quantity< boost::
            units::si::capacitance >;
134
135
136
137     timing_data & m_timing;
138     graph_and_topology * m_topology;
139     const entity_system::vector_property< interconnection::
            packed_rc_tree > & m_rc_trees;
140     MergeStrategy m_merge;
```

```
141
142     SlewType compute_slew(lemon::ListDigraph::Node node,
            CapacitanceType load) const {
143         SlewType worst_slew = MergeStrategy::best();
144         if(lemon::countInArcs(m_topology->g.G(), node) == 0)
                // PI without driver
145             return m_timing.nodes.slew(node);
146         switch(m_topology->g.node_edge(node))
147         {
148         case edges::RISE:
149             for(lemon::ListDigraph::InArcIt it(m_topology->g
                    .G(), node); it != lemon::INVALID; ++it)
150             {
151                 auto tarc = m_topology->g.edge_entity(it) ;
152                 worst_slew = m_merge(worst_slew, m_timing.
                        lib.timing_arc_rise_slew(tarc).compute(
                        load, m_timing.nodes.slew(m_topology->g.
                        edge_source(it))));
153             }
154             break;
155         case edges::FALL:
156             for(lemon::ListDigraph::InArcIt it(m_topology->g
                    .G(), node); it != lemon::INVALID; ++it)
157             {
158                 auto tarc = m_topology->g.edge_entity(it) ;
159                 worst_slew = m_merge(worst_slew, m_timing.
                        lib.timing_arc_fall_slew(tarc).compute(
                        load, m_timing.nodes.slew(m_topology->g.
                        edge_source(it))));
160             }
161             break;
162         }
163         return worst_slew;
164     }
165
166 public:
167     generic_sta( timing_data & timing, graph_and_topology &
            topology, const entity_system::vector_property<
            interconnection::packed_rc_tree > & rc_trees) :
168         m_timing(timing),
169         m_topology(&topology),
170         m_rc_trees(rc_trees)
171     {
172
173
174
175
176     }
177
178
179     void topology(graph_and_topology & topology)
180     {
```

```
181         m_topology = &topology;
182     }
183
184
185
186     void set_constraints(const design_constraints & dc)
187     {
188
189         using namespace boost::units;
190         using namespace boost::units::si;
191
192         m_timing.nodes.arrival( m_topology->g.rise_node(
                m_topology->netlist.pin_by_name(dc.clock.
                port_name)), 0.0*seconds );
193         m_timing.nodes.arrival( m_topology->g.fall_node(
                m_topology->netlist.pin_by_name(dc.clock.
                port_name)), 0.0*seconds );
194
195         for(auto & i : dc.input_delays)
196         {
197             //auto pin = m_topology->netlist.pin_by_name(i.
                    port_name);
198             auto pin = m_topology->netlist.pin_by_name(i.
                    port_name + "_driver");
199             m_timing.nodes.arrival( m_topology->g.rise_node(
                    pin), quantity<si::time>(i.delay*pico*
                    seconds) );
200             m_timing.nodes.arrival( m_topology->g.fall_node(
                    pin), quantity<si::time>(i.delay*pico*
                    seconds) );
201         }
202
203         for(auto & i : dc.input_drivers)
204         {
205             //auto pin = m_topology->netlist.pin_by_name(i.
                    port_name);
206             auto pin = m_topology->netlist.pin_by_name(i.
                    port_name + "_driver");
207             m_timing.nodes.slew( m_topology->g.rise_node(pin
                    ), quantity<si::time>(i.slew_rise*pico*
                    seconds) );
208             m_timing.nodes.slew( m_topology->g.fall_node(pin
                    ), quantity<si::time>(i.slew_fall*pico*
                    seconds) );
209         }
210
211
212         for(lemon::ListDigraph::NodeIt node(m_topology->g.G
                ()); node != lemon::INVALID; ++node)
213         {
214             if(m_timing.lib.pin_clock_input(m_topology->
                    netlist.pin_std_cell(m_topology->g.pin(node)
```

```
                ) ))
215             m_timing.nodes.required( node, MergeStrategy
                    ::worst() );
216         else if(lemon::countOutArcs(m_topology->g.G(),
                node) == 0 )
217             m_timing.nodes.required( node, m_merge(
                    quantity<si::time>(0.0*seconds),
                    quantity<si::time>(dc.clock.period *
                    pico* seconds)) );
218     }
219
220 }
221
222
223 SlewType rise_arrival(const entity_system::entity pin)
        const
224 {
225     return m_timing.nodes.arrival(m_topology->g.
            rise_node(pin));
226 }
227 SlewType fall_arrival(const entity_system::entity pin)
        const
228 {
229     return m_timing.nodes.arrival(m_topology->g.
            fall_node(pin));
230 }
231
232 SlewType rise_slew(const entity_system::entity pin)
        const
233 {
234     return m_timing.nodes.slew(m_topology->g.rise_node(
            pin));
235 }
236 SlewType fall_slew(const entity_system::entity pin)
        const
237 {
238     return m_timing.nodes.slew(m_topology->g.fall_node(
            pin));
239 }
240
241 SlewType rise_slack(const entity_system::entity pin)
        const
242 {
243     auto node = m_topology->g.rise_node(pin);
244     return MergeStrategy::slack_signal()*(m_timing.nodes
            .required(node)-m_timing.nodes.arrival(node));
245 }
246 SlewType fall_slack(const entity_system::entity pin)
        const
247 {
248     auto node = m_topology->g.fall_node(pin);
```

```
249             return MergeStrategy::slack_signal()*(m_timing.nodes
                    .required(node)-m_timing.nodes.arrival(node));
250         }
251
252         void update_ats() {
253         #pragma omp parallel
254         for(auto & level : m_topology->levels){
255           #pragma omp for
256           for(std::size_t i = 0; i < level.size(); ++i){ //
                  paralell
257             auto node = level[i];
258             if(lemon::countInArcs(m_topology->g.G(), node) !=
                    0){
259               auto pin = m_topology->g.pin(node);
260               auto net = m_topology->netlist.pin_net(pin);
261               auto & tree = m_rc_trees[m_topology->netlist.
                      net_system().lookup(net)];
262
263               std::vector< SlewType > slews(tree.node_count())
                      ;
264               std::vector< SlewType > delays(tree.node_count()
                      );
265               std::vector< CapacitanceType > ceffs(tree.
                      node_count());
266               WireDelayModel calculator;
267               calculator.delay_map(delays);
268               calculator.slew_map(slews);
269               calculator.ceff_map(ceffs);
270               std::function<SlewType(CapacitanceType)>
                      s_calculator = std::bind(&generic_sta::
                      compute_slew, this, node, std::placeholders
                      ::_1);
271
272               CapacitanceType load = calculator.simulate(
                      s_calculator, tree);
273
274               m_timing.nodes.load(node, load);
275               m_timing.nodes.slew(node, slews[0]);
276
277               SlewType worst_arrival = MergeStrategy::best();
278               switch(m_topology->g.node_edge(node)){
279                 case edges::RISE:
280                   for(lemon::ListDigraph::InArcIt it(
                          m_topology->g.G(), node); it != lemon::
                          INVALID; ++it)
281                   {
282                     auto tarc = m_topology->g.edge_entity(it
                            );
283                     auto edge_source = m_topology->g.
                            edge_source(it);
284                     auto arc_delay = m_timing.lib.
                            timing_arc_rise_delay(tarc).compute(
```

```
                        load, m_timing.nodes.slew(
                            edge_source));
285                     auto arc_slew = m_timing.lib.
                            timing_arc_rise_slew(tarc).compute(
                            load, m_timing.nodes.slew(
                            edge_source));
286                     m_timing.arcs.delay(it, arc_delay);
287                     m_timing.arcs.slew(it, arc_slew);
288                     worst_arrival = m_merge(worst_arrival,
                            m_timing.nodes.arrival(edge_source)
                            + arc_delay);
289                   }
290                 break;
291                 case edges::FALL:
292                   for(lemon::ListDigraph::InArcIt it(
                            m_topology->g.G(), node); it != lemon::
                            INVALID; ++it)
293                   {
294                     auto tarc = m_topology->g.edge_entity(it
                            );
295                     auto edge_source = m_topology->g.
                            edge_source(it);
296                     auto arc_delay = m_timing.lib.
                            timing_arc_fall_delay(tarc).compute(
                            load, m_timing.nodes.slew(
                            edge_source));
297                     auto arc_slew = m_timing.lib.
                            timing_arc_fall_slew(tarc).compute(
                            load, m_timing.nodes.slew(
                            edge_source));
298                     m_timing.arcs.delay(it, arc_delay);
299                     m_timing.arcs.slew(it, arc_slew);
300                     worst_arrival = m_merge(worst_arrival,
                            m_timing.nodes.arrival(edge_source)
                            + arc_delay);
301                   }
302                 break;
303               }
304               m_timing.nodes.arrival(node, worst_arrival);
305               for(lemon::ListDigraph::OutArcIt arc(m_topology
                      ->g.G(), node); arc != lemon::INVALID; ++arc
                      )
306               {
307                 auto arc_target = m_topology->g.edge_target(
                        arc);
308                 auto target_pin = m_topology->g.pin(
                        arc_target);
309                 auto target_capacitor = tree.tap(m_topology
                        ->netlist.pin_name(target_pin));
310                 m_timing.arcs.slew(arc, slews[
                        target_capacitor]);
```

```
311                 m_timing.arcs.delay(arc, delays[
                        target_capacitor]);
312                 m_timing.nodes.slew(arc_target, m_timing.
                        arcs.slew(arc));
313                 m_timing.nodes.arrival(arc_target, m_timing.
                        nodes.arrival(node) + m_timing.arcs.
                        delay(arc));
314             }
315         }//end if in_arcs != 0
316       }//end parallel level
317       }//end outside for
318     }
319     void update_rts_parallel(){
320         #pragma omp parallel for
321         for(auto node_it = m_topology->sorted.rbegin();
                node_it > m_topology->sorted.rend(); ++node_it)
322         {
323             auto node = *node_it;
324             if(lemon::countOutArcs(m_topology->g.G(), node)
                    > 0)
325             {
326                 SlewType required = MergeStrategy::worst();
327                 for(lemon::ListDigraph::OutArcIt arc(
                        m_topology->g.G(), node); arc != lemon::
                        INVALID; ++arc)
328                     required = m_merge.inverted(required,
                            m_timing.nodes.required(m_topology->
                            g.edge_target(arc))-m_timing.arcs.
                            delay(arc));
329                 m_timing.nodes.required(node, required);
330             }
331         }
332     }
333
334
335     void update_rts() {
336         for(auto node_it = m_topology->sorted.rbegin();
                node_it != m_topology->sorted.rend(); ++node_it)
337         {
338             auto node = *node_it;
339             if(lemon::countOutArcs(m_topology->g.G(), node)
                    > 0)
340             {
341                 SlewType required = MergeStrategy::worst();
342                 for(lemon::ListDigraph::OutArcIt arc(
                        m_topology->g.G(), node); arc != lemon::
                        INVALID; ++arc)
343                     required = m_merge.inverted(required,
                            m_timing.nodes.required(m_topology->
                            g.edge_target(arc))-m_timing.arcs.
                            delay(arc));
344                 m_timing.nodes.required(node, required);
```

```
345                 }
346             }
347         }
348
349
350         lemon::Path<lemon::ListDigraph> critical_path() const {
351             lemon::Path<lemon::ListDigraph> cp;
352             SlewType worst_slack = std::numeric_limits<SlewType
                    >::infinity();
353             lemon::ListDigraph::Node worst_PO;
354             for(auto node_it = m_topology->sorted.rbegin();
                    node_it != m_topology->sorted.rend(); ++node_it)
355             {
356                 auto node = *node_it;
357                 if(lemon::countOutArcs(m_topology->g.G(), node)
                        == 0)
358                 {
359                     SlewType current_PO_slack = MergeStrategy::
                            slack_signal()*(m_timing.nodes.required(
                            node)-m_timing.nodes.arrival(node));
360                     if(current_PO_slack < worst_slack)
361                     {
362                         worst_slack = current_PO_slack;
363                         worst_PO = node;
364                     }
365                 }
366             }
367             lemon::ListDigraph::Node current_node = worst_PO;
368             lemon::ListDigraph::Node next_node = current_node;
369             while(next_node != lemon::INVALID)
370             {
371                 current_node = next_node;
372                 next_node = lemon::INVALID;
373                 lemon::ListDigraph::Arc worst_arc = lemon::
                        INVALID;
374                 SlewType worst_slack_input = std::numeric_limits
                        <SlewType>::infinity();
375                 for(lemon::ListDigraph::InArcIt in(m_topology->g
                        .G(), current_node); in != lemon::INVALID;
                        ++in)
376                 {
377                     auto source = m_topology->g.G().source(in);
378                     SlewType slack = MergeStrategy::slack_signal
                            ()*(m_timing.nodes.required(source)-
                            m_timing.nodes.arrival(source));
379                     if(slack < worst_slack_input)
380                     {
381                         worst_slack_input = slack;
382                         worst_arc = in;
383                         next_node = source;
384                     }
385                 }
```

```
386                 if(worst_arc != lemon::INVALID)
387                     cp.addFront(worst_arc);
388             }
389             return cp;
390         }
391
392
393 };
394
395 }
396 }
397
398
399 #endif // OPHIDIAN_TIMING_GENERIC_STA_H
```

Listing C.1 – generic_sta.h

```
1  /*
2   * Copyright 2016 Ophidian
3  Licensed to the Apache Software Foundation (ASF) under one
4  or more contributor license agreements.  See the NOTICE file
5  distributed with this work for additional information
6  regarding copyright ownership.  The ASF licenses this file
7  to you under the Apache License, Version 2.0 (the
8  "License"); you may not use this file except in compliance
9  with the License.  You may obtain a copy of the License at
10
11   http://www.apache.org/licenses/LICENSE-2.0
12
13 Unless required by applicable law or agreed to in writing,
14 software distributed under the License is distributed on an
15 "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
16 KIND, either express or implied.  See the License for the
17 specific language governing permissions and limitations
18 under the License.
19  */
20
21 #include "generic_sta.h"
22
23 namespace ophidian {
24 namespace timing {
25
26 void test_calculator:: compute_tests()
27 {
28     auto tests = topology.g.tests();
29     std::size_t i;
30     using DelayType = boost::units::quantity< boost::units::
          si::time >;
31 #pragma omp parallel for
32     for(i = 0; i < tests.size(); ++i) // parallel
33     {
34         const test & t = tests[i];
```

```
35          auto pin_ck = topology.g.pin(t.ck);
36          auto pin_d = topology.g.pin(t.d);
37          DelayType setup, hold;
38          switch(topology.g.node_edge(t.d))
39          {
40          case edges::RISE:
41              setup = late.lib.setup_rise(t.tarc).compute(
                    early.nodes.slew(t.ck), late.nodes.slew(t.d)
                    );
42              hold = early.lib.hold_rise(t.tarc).compute(late.
                    nodes.slew(t.ck), early.nodes.slew(t.d));
43              break;
44          case edges::FALL:
45              setup = late.lib.setup_fall(t.tarc).compute(
                    early.nodes.slew(t.ck), late.nodes.slew(t.d)
                    );
46              hold = early.lib.hold_fall(t.tarc).compute(late.
                    nodes.slew(t.ck), early.nodes.slew(t.d));
47              break;
48          }
49          const DelayType e_ck = early.nodes.arrival(topology.
                g.rise_node(pin_ck));
50          const DelayType l_ck = late.nodes.arrival(topology.g
                .rise_node(pin_ck));
51          late.nodes.required(t.d, e_ck-setup+clock_period);
52          early.nodes.required(t.d, l_ck+hold);
53      }
54  }
55
56  graph_and_topology::graph_and_topology(const graph &G, const
        netlist::netlist &netlist, const library &lib):
57      g(G),
58      netlist(netlist),
59      sorted(g.nodes_count()),
60      sorted_drivers(g.nodes_count()){
61
62      using GraphType = lemon::ListDigraph;
63
64      GraphType::NodeMap<int> order(g.G());
65      lemon::topologicalSort(g.G(), order);
66
67
68      std::vector<GraphType::Node> sorted_nodes(g.nodes_count
            ());
69
70      GraphType::NodeMap<int> level(g.G());
71
72      for(GraphType::NodeIt it(g.G()); it != lemon::INVALID;
            ++it)
73      {
74          level[it] = std::numeric_limits<int>::max();
75          sorted[ order[it] ] = it;
```

```
76          sorted_drivers[ order[it] ] = it;
77          if(lemon::countInArcs(g.G(), it) == 0)
78              level[it] = 0;
79      }
80
81      int num_levels = 0;
82      for(auto node : sorted)
83      {
84          if(lemon::countInArcs(g.G(), node) > 0)
85          {
86              int max_level = std::numeric_limits<int>::min();
87              for(GraphType::InArcIt arc(g.G(), node); arc !=
                    lemon::INVALID; ++arc)
88                  max_level = std::max(max_level, level[g.
                        edge_source(arc)]);
89              level[node] = max_level + 1;
90              num_levels = std::max(num_levels, max_level+1);
91          }
92      }
93
94      levels.resize(num_levels+1);
95
96      for(auto node : sorted)
97      {
98          if(lib.pin_direction(netlist.pin_std_cell(g.pin(node
                ))) == standard_cell::pin_directions::OUTPUT)
99              levels[level[node]].push_back(node);
100     }
101
102     auto beg = std::remove_if(levels.begin(), levels.end(),
            [this](std::vector<lemon::ListDigraph::Node> & vec)
            ->bool{
103             return vec.empty();
104 });
105     levels.erase(beg, levels.end());
106
107
108     auto begin = std::remove_if(
109                 sorted_drivers.begin(),
110                 sorted_drivers.end(),
111                 [this, lib, netlist](GraphType::Node node)->
                        bool {
112             return lib.pin_direction(netlist.pin_std_cell(g.
                    pin(node))) != standard_cell::pin_directions
                    ::OUTPUT;
113 });
114
115     sorted_drivers.erase(begin, sorted_drivers.end());
116 #ifndef NDEBUG
117     std::for_each(sorted_drivers.begin(), sorted_drivers.end
            (), [this, lib, netlist](GraphType::Node node){
118         assert(lib.pin_direction(netlist.pin_std_cell(g.pin(
```

```
                node))) == standard_cell::pin_directions::OUTPUT
                );
119         });
120         std::for_each(levels.begin(), levels.end(), [this](std::
                vector<lemon::ListDigraph::Node> & vec)->bool{
121             assert(!vec.empty());
122         });
123 #endif
124 }
125
126 }
127 }
```

Listing C.2 – generic_sta.cpp

```
1  /*
2   * Copyright 2016 Ophidian
3  Licensed to the Apache Software Foundation (ASF) under one
4  or more contributor license agreements.  See the NOTICE file
5  distributed with this work for additional information
6  regarding copyright ownership.  The ASF licenses this file
7  to you under the Apache License, Version 2.0 (the
8  "License"); you may not use this file except in compliance
9  with the License.  You may obtain a copy of the License at
10
11   http://www.apache.org/licenses/LICENSE-2.0
12
13 Unless required by applicable law or agreed to in writing,
14 software distributed under the License is distributed on an
15 "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
16 KIND, either express or implied.  See the License for the
17 specific language governing permissions and limitations
18 under the License.
19  */
20
21 #ifndef OPHIDIAN_TIMING_STATIC_TIMING_ANALYSIS_H
22 #define OPHIDIAN_TIMING_STATIC_TIMING_ANALYSIS_H
23
24 #include "generic_sta.h"
25 #include "endpoints.h"
26 #include <chrono>
27
28 namespace ophidian {
29 namespace timing {
30
31
32 using TimeType = boost::units::quantity< boost::units::si::
        time > ;
33 using Cell = entity_system::entity;
34 using Net = entity_system::entity;
35 using Pin = entity_system::entity;
36
```

```
37
38
39  class static_timing_analysis
40  {
41      const timing::graph * m_timing_graph;
42      const entity_system::vector_property< interconnection::
            packed_rc_tree > * m_rc_trees;
43      const library * m_late_lib;
44      const library * m_early_lib;
45      const netlist::netlist * m_netlist;
46      design_constraints m_dc;
47
48
49      // lazy pointers
50      std::unique_ptr<timing_data> m_late;
51      std::unique_ptr<timing_data> m_early;
52      std::unique_ptr<graph_and_topology> m_topology;
53      std::unique_ptr<generic_sta<
            lumped_capacitance_wire_model, pessimistic> >
            m_late_sta;
54      std::unique_ptr<generic_sta<
            lumped_capacitance_wire_model, optimistic> >
            m_early_sta;
55      std::unique_ptr<test_calculator> m_test;
56      endpoints m_endpoints;
57      TimeType m_lwns;
58      TimeType m_ewns;
59      TimeType m_ltns;
60      TimeType m_etns;
61
62      void init_timing_data();
63      void propagate_ats();
64      void propagate_rts();
65      void update_wns_and_tns();
66      bool has_timing_data() const {
67          assert(m_rc_trees);
68          assert(m_timing_graph);
69          assert(m_late_lib && m_early_lib);
70          assert(m_netlist);
71          return m_late_sta && m_early_sta;
72      }
73  public:
74      static_timing_analysis();
75      void graph(const timing::graph& g);
76      void rc_trees(const entity_system::vector_property<
            interconnection::packed_rc_tree> &trees);
77      void late_lib(const library& lib);
78      void early_lib(const library& lib);
79      void netlist(const netlist::netlist & netlist);
80      void set_constraints(const design_constraints & dc);
81
82      void update_timing();
```

```
83
84
85      TimeType late_wns() const {
86          return m_lwns;
87      }
88      TimeType early_wns() const{
89          return m_ewns;
90      }
91      TimeType late_tns() const {
92          return m_ltns;
93      }
94      TimeType early_tns() const{
95          return m_etns;
96      }
97
98      TimeType early_rise_slack(Pin p) const {
99          return m_early_sta->rise_slack(p);
100     }
101     TimeType early_fall_slack(Pin p) const {
102         return m_early_sta->fall_slack(p);
103     }
104     TimeType late_rise_slack(Pin p) const {
105         return m_late_sta->rise_slack(p);
106     }
107     TimeType late_fall_slack(Pin p) const {
108         return m_late_sta->fall_slack(p);
109     }
110
111     TimeType early_rise_arrival(Pin p) const {
112         return m_early_sta->rise_arrival(p);
113     }
114     TimeType early_fall_arrival(Pin p) const {
115         return m_early_sta->fall_arrival(p);
116     }
117     TimeType late_rise_arrival(Pin p) const {
118         return m_late_sta->rise_arrival(p);
119     }
120     TimeType late_fall_arrival(Pin p) const {
121         return m_late_sta->fall_arrival(p);
122     }
123
124     TimeType early_rise_slew(Pin p) const {
125         return m_early_sta->rise_slew(p);
126     }
127     TimeType early_fall_slew(Pin p) const {
128         return m_early_sta->fall_slew(p);
129     }
130     TimeType late_rise_slew(Pin p) const {
131         return m_late_sta->rise_slew(p);
132     }
133     TimeType late_fall_slew(Pin p) const {
134         return m_late_sta->fall_slew(p);
```

```
135     }
136
137     const endpoints & timing_endpoints() const {
138         return m_endpoints;
139     }
140
141 };
142
143 }
144 }
145
146
147 #endif // OPHIDIAN_TIMING_STATIC_TIMING_ANALYSIS_H
```

Listing C.3 – static_timing_analysis.h

```
1  /*
2   * Copyright 2016 Ophidian
3  Licensed to the Apache Software Foundation (ASF) under one
4  or more contributor license agreements.  See the NOTICE file
5  distributed with this work for additional information
6  regarding copyright ownership.  The ASF licenses this file
7  to you under the Apache License, Version 2.0 (the
8  "License"); you may not use this file except in compliance
9  with the License.  You may obtain a copy of the License at
10
11   http://www.apache.org/licenses/LICENSE-2.0
12
13 Unless required by applicable law or agreed to in writing,
14 software distributed under the License is distributed on an
15 "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
16 KIND, either express or implied.  See the License for the
17 specific language governing permissions and limitations
18 under the License.
19  */
20
21 #include "static_timing_analysis.h"
22 #include "graph_builder.h"
23 #include "tns.h"
24
25 #include "wns.h"
26
27 namespace ophidian {
28 namespace timing {
29
30 void static_timing_analysis::init_timing_data()
31 {
32     std::cout<<"setting librarys"<<std::endl;
33     m_late.reset(new timing::timing_data(*m_late_lib, *
        m_timing_graph));
34     m_early.reset(new timing::timing_data(*m_early_lib, *
        m_timing_graph));
```

```
35      std::cout<<"setting topology"<<std::endl;
36      m_topology.reset(new timing::graph_and_topology(*
            m_timing_graph, *m_netlist, *m_late_lib));
37      std::cout<<"setting stas"<<std::endl;
38      m_late_sta.reset(new timing::generic_sta<timing::
            lumped_capacitance_wire_model, timing::pessimistic
            >(*m_late, *m_topology, *m_rc_trees));
39      m_early_sta.reset(new timing::generic_sta<timing::
            lumped_capacitance_wire_model, timing::optimistic>(*
            m_early, *m_topology, *m_rc_trees));
40      std::cout<<"setting m_test"<<std::endl;
41      m_test.reset(new timing::test_calculator{*m_topology, *
            m_early, *m_late, TimeType(m_dc.clock.period*boost::
            units::si::pico*boost::units::si::seconds)});
42      std::cout<<"setting endpoints"<<std::endl;
43      m_endpoints = timing::endpoints(*m_netlist);
44      std::cout<<"setting constraints"<<std::endl;
45      m_late_sta->set_constraints(m_dc);
46      m_early_sta->set_constraints(m_dc);
47  }
48
49  void static_timing_analysis::propagate_ats()
50  {
51      m_late_sta->update_ats();
52      m_early_sta->update_ats();
53  }
54
55  void static_timing_analysis::propagate_rts()
56  {
57      m_late_sta->update_rts_parallel();
58      m_early_sta->update_rts_parallel();
59  }
60
61  void static_timing_analysis::update_wns_and_tns()
62  {
63      m_lwns = timing::wns(m_endpoints, *m_late_sta).value();
64      m_ewns = timing::wns(m_endpoints, *m_early_sta).value();
65      m_ltns = timing::tns(m_endpoints, *m_late_sta).value();
66      m_etns = timing::tns(m_endpoints, *m_early_sta).value();
67
68  }
69
70  static_timing_analysis::static_timing_analysis() :
71      m_timing_graph(nullptr),
72      m_rc_trees(nullptr)
73  {
74
75  }
76
77  void static_timing_analysis::update_timing()
78  {
79      std::cout<<"setting timing data"<<std::endl;
```

```
80      if(!has_timing_data())
81          init_timing_data();
82      std::cout<<"updating ats"<<std::endl;
83      std::chrono::steady_clock::time_point ats_1(std::chrono
            ::steady_clock::now());
84      propagate_ats();
85      std::chrono::steady_clock::time_point ats_2(std::chrono
            ::steady_clock::now());
86      std::cout <<"To propagate ats: "<< std::chrono::
            duration_cast<std::chrono::duration<double>>(ats_2 -
             ats_1).count()<<std::endl;
87      std::cout<<"computing tests"<<std::endl;
88      std::chrono::steady_clock::time_point t_1(std::chrono::
            steady_clock::now());
89      m_test->compute_tests();
90      std::chrono::steady_clock::time_point t_2(std::chrono::
            steady_clock::now());
91      std::cout <<"To compute tests: "<< std::chrono::
            duration_cast<std::chrono::duration<double>>(t_2 -
            t_1).count()<<std::endl;
92      std::cout<<"updating rts"<<std::endl;
93      std::chrono::steady_clock::time_point rt_1(std::chrono::
            steady_clock::now());
94      propagate_rts();
95      std::chrono::steady_clock::time_point rt_2(std::chrono::
            steady_clock::now());
96      std::cout <<"To update rts: "<< std::chrono::
            duration_cast<std::chrono::duration<double>>(rt_2 -
            rt_1).count()<<std::endl;
97
98      update_wns_and_tns();
99  }
100
101
102 void static_timing_analysis::graph(const ophidian::timing::
        graph &g)
103 {
104     m_timing_graph = &g;
105 }
106
107 void static_timing_analysis::rc_trees(const entity_system::
        vector_property<interconnection::packed_rc_tree> &trees)
108 {
109     m_rc_trees = &trees;
110 }
111
112 void static_timing_analysis::late_lib(const library &lib)
113 {
114     m_late_lib = &lib;
115 }
116
117 void static_timing_analysis::early_lib(const library &lib)
```

```
118 {
119     m_early_lib = &lib;
120 }
121
122 void static_timing_analysis::netlist(const netlist::netlist
        &netlist)
123 {
124     m_netlist = &netlist;
125 }
126
127 void static_timing_analysis::set_constraints(const
        design_constraints &dc)
128 {
129     m_dc = dc;
130 }
131
132 }
133 }
```

Listing C.4 – static_timing_analysis.cpp